DrawnApart: A Device Identification and Spoofing Detection Technique based on Remote GPU Fingerprinting

TOMER LAOR*, Ben-Gurion University of the Negev, Israel
NAIF MEHANNA*, Univ. Lille, CNRS, Inria, France
ANTONIN DUREY, Univ. Lille, CNRS, Inria, France
VITALY DYADYUK, Ben-Gurion University of the Negev, Israel
PIERRE LAPERDRIX, Univ. Lille, CNRS, Inria, France
CLÉMENTINE MAURICE, Univ. Lille, CNRS, Inria, France
YOSSI OREN, Ben-Gurion University of the Negev, Israel
ROMAIN ROUVOY, Univ. Lille, CNRS, Inria, France
WALTER RUDAMETKIN, Univ. Lille, CNRS, Inria, France
YUVAL YAROM, Ruhr University Bochum, Germany

Browser fingerprinting aims to identify users or their devices, through scripts that execute in the users' browser and collect information on software or hardware characteristics. It is used to track users or as an additional means of identification to improve security. Fingerprinting techniques have one significant limitation: they are unable to track individual users for an extended duration. This happens because browser fingerprints evolve over time, and these evolutions ultimately cause a fingerprint to be confused with those from other devices sharing similar hardware and software.

Our technique, DrawnApart, advances the field of browser fingerprinting in three significant ways. First, it is the pioneering work that investigates and exploits manufacturing variances in identical GPUs in the context of privacy protection. Second, it provides a robust method for differentiating between devices with identical hardware and software configurations, achieving practical accuracy improvements in real-world scenarios. In a setting with diverse hardware and software configurations, DrawnApart is most effective when combined with traditional fingerprinting techniques such as FP-Stalker [74]. Third, DrawnApart introduces a capability to verify the GPU renderer string, enabling the detection of spoofing attempts aimed at bypassing two-factor authentication by mimicking the victim's device attributes.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**.

*Both authors contributed equally to this research.

Authors' addresses: Tomer Laor, tomerlao@post.bgu.ac.il, Ben-Gurion University of the Negev, Israel; Naif Mehanna, naif.mehanna@univ-lille.fr, Univ. Lille, CNRS, Inria, France; Antonin Durey, antonin.durey@univ-lille.fr, Univ. Lille, CNRS, Inria, France; Vitaly Dyadyuk, vitalyd@post.bgu.ac.il, Ben-Gurion University of the Negev, Israel; Pierre Laperdrix, pierre.laperdrix@univ-lille.fr, Univ. Lille, CNRS, Inria, France; Clémentine Maurice, clementine.maurice@inria.fr, Univ. Lille, CNRS, Inria, France; Yossi Oren, yos@bgu.ac.il, Ben-Gurion University of the Negev, Israel; Romain Rouvoy, romain.rouvoy@univ-lille.fr, Univ. Lille, CNRS, Inria, France; Walter Rudametkin, walter.rudametkin@univ-lille.fr, Univ. Lille, CNRS, Inria, France; Yuval Yarom, yuval.yarom@rub.de, Ruhr University Bochum, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Association for Computing Machinery.
Manuscript submitted to ACM

Additional Key Words and Phrases: Device Fingerprinting, Microarchitectural Fingerprinting, WebGL, Spoofing Detection, Web Security, Privacy, Side-Channel Attacks

ACM Reference Format:
Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. 2026. DrawnApart: A Device Identification and Spoofing Detection Technique based on Remote GPU Fingerprinting. 1, 1 (July 2026), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Privacy is dignity. It is a human right. In the domain of web browsing, the right to privacy should prevent websites from tracking user browsing activity without consent. This is the case in particular for cross-site tracking, in which website owners collude to build browsing profiles spanning multiple websites over extended periods of time. Unfortunately for users, the right to privacy conflicts with business interests. Website owners are highly interested in tracking users for the purpose of showing them ads they are more likely to click on, or to recommend products they are more likely to purchase.

We focus on the common scenario where identifying a browser is equivalent to tracking a user. The traditional way to track users is with cookies, small files that are stored by the browser at the request of the website, and forwarded to the website on demand [45]. Recent regulations restrict and supervise the acquisition of private data by websites [3, 27], and in particular require that users consent to the use of cookies. Furthermore, in an effort to protect users' privacy and curb tracking, modern browsers restrict cookie-based tracking, especially *third-party trackers* that attempt to track users across multiple unrelated websites.

To overcome the limitations of cookies, less scrupulous websites often resort to an approach called *browser fingerprinting*. To fingerprint a browser, the website provides a script that queries the browser's software and hardware configuration to collect attributes, such as the browser's version, OS, timezone, screen, language, list of fonts, or even the way the browser renders text and graphics. The diversity of configurations allows websites to discriminate devices and, hence, to track users, without the use of cookies [49], even in a collection spanning millions of fingerprints [37]. Surveying the Internet demonstrates that browser fingerprinting techniques are prevalent and used by many websites, no matter their category or ranking [31, 33, 60]. Browser fingerprinting is sometimes put to constructive use — recent studies show that the login pages of multiple high-profile websites, including banking and credit card company websites, use browser fingerprinting to waive two-factor authentication (2FA) for returning users. Adversaries, however, may abuse this mechanism to bypass 2FA [52] by stealing the browser fingerprint along with the user's credentials.

A significant difficulty of fingerprint-based tracking is that browser fingerprints evolve. As shown by Vastel et al. [74], fingerprints change frequently, sometimes multiple times per day, due to software updates and configuration changes. To track a user, an adversary must link fingerprint evolutions into a single coherent chain. This process is made difficult by the existence of devices with identical hardware and software configurations. It is difficult for an adversary to correctly link a fingerprint if there is a set of identical devices to which it might belong. This limits the adversary's tracking duration. In Vastel et al.'s evaluation over a dataset of nearly 100,000 fingerprints collected from 1,905 distinct browser instances, with a wide variety of fingerprinting attributes, their state-of-the-art machine learning technique was able to deliver a median tracking time of less than two months.

Another challenge for browser fingerprinting is the ease with which attackers can spoof these fingerprints. An adversary can first record the browser fingerprint of a victim, and then use this fingerprint to impersonate the victim in future visits [52]. This is a rising concern, since a large amount of phishing pages have already been observed recording

browser fingerprints [65], indicating that adversaries may be planning to use them together with the victim's stolen credentials to bypass 2FA and impersonate users [52].

In this work, we bring a new insight to the challenge of browser fingerprinting identical computers, by observing that even nominally identical hardware devices have slight differences induced by their manufacturing process. These manufacturing variations are shown to enable the extraction of unique and robust fingerprints from a variety of devices, both large and small, in other settings [38, 71]. If an adversary were able to extract such a hardware fingerprint from the user's device, it would significantly extend the adversary's ability to track them. Conversely, a defender can detect spoofing attempts by looking for incongruities between this hardware fingerprint and other fingerprint attributes collected by other methods.

Challenges. Extracting a hardware fingerprint from a browser is, however, far from trivial, as it requires adapting hardware-induced timing measurements, which are traditionally performed in native and highly controlled environments, to the Web execution model. In practice, the attacker is confined to unprivileged JavaScript code and WebGL graphics primitives, with no control over GPU scheduling, runtime configuration, or concurrent system activity. The browser sandbox further restricts timing resolution, while the interaction time with a user is inherently limited, often to only a few seconds, which severely constrains the duration and structure of measurements. As a result, classical fingerprinting and machine-learning pipelines that rely on long or repeated training phases are ineffective in this setting. Thus, in this paper we raise the following question:

Can browser fingerprinting work on devices with identical hardware and software configurations?

Our Contribution. We claim this is possible, and we assess this claim with DrawnApart, a technique that measures small differences among the *Execution Units* (EUs) that make up a modern *Graphics Processing Unit* (GPU). By fingerprinting the GPU stack, DrawnApart can tell apart devices with nominally identical configurations, both in the lab and in the wild. In a nutshell, to create a fingerprint, DrawnApart generates a sequence of rendering tasks, each targeting different EUs. It times each rendering task, creating a fingerprint trace. This trace is transformed by a deep learning network into an *embedding vector* that describes it succinctly and points the adversary towards the specific device that generated it. We emphasize that identifying the precise physical causes of these timing differences is out of scope; our approach leverages their existence and stability, rather than their underlying origin.

We evaluate DrawnApart in three main scenarios. First, to validate the method's ability to distinguish nominally identical configurations, we perform a series of controlled experiments under lab conditions. We experiment with multiple sets of identical devices from vendors including Intel, Apple, Nvidia and Samsung, and demonstrate that DrawnApart consistently improves identification of these nominally identical devices, achieving high identification accuracy in multiple hardware configurations, even though state-of-the-art browser-based fingerprinting methods cannot tell them apart. Second, to show that DrawnApart affects user privacy, we integrated it into Vastel et al.'s state-of-the-art fingerprinting algorithm from IEEE S&P 2018 [74], which uses machine learning to link browser fingerprint evolutions. Using this integrated system, we conducted an in-the-wild evaluation using a crowd-sourced dataset of over 2,500 machines featuring 1,605 distinct GPU configurations. We show that the median tracking duration is improved by up to 66.66% once we add the DrawnApart fingerprint.

Third, to show how DrawnApart can be used for spoofing prevention, we demonstrate that DrawnApart can be used to selectively waive two-factor authentication (2FA) in some settings to maintain usability while protecting from spoofing, contrasting it with the current method, which relies on matching deterministic attributes between consecutive

visits and was shown to be susceptible to spoofing [52]. We show that DrawnApart can waive 2FA for 19.4% of benign visits while detecting 99% of spoofing attempts.

In summary, this paper makes the following contributions:

- We design and implement DrawnApart[1], a GPU fingerprinting technique based on the relative speed of EUs, that observes minute differences between GPUs (Section 3).
- We investigate the performance of our fingerprinting technique with multiple sets of identical devices, demonstrating that it can tell apart devices with identical hardware and software configurations (Section 5).
- We integrate DrawnApart into Vastel et al.'s fingerprinting algorithm and show, through a large-scale crowd-sourced experiment with over 2,500 unique devices and almost 371,000 fingerprints, that DrawnApart delivers considerable gains to the tracking accuracy of this state-of-the-art approach (Section 6).
- We demonstrate that DrawnApart can be used to detect spoofing attempts, mitigating 2FA bypass on 99% of the cases while maintaining low user friction (Section 7).
- We suggest possible countermeasures against our fingerprinting technique, and discuss their advantages and drawbacks (Section 8.2).

2 BACKGROUND

2.1 Browser Fingerprinting

Mowery et al. [56] discuss fingerprinting on the Web. As they state, fingerprinting can be applied constructively or destructively. An example of constructive use of fingerprints would be to identify fraudulent users trying to log in while masquerading as legitimate users. Browser fingerprinting can be used to detect bots [23, 42, 75], or support authentication, where the fingerprint is used in addition to a traditional authentication mechanism [16, 47]. A destructive use might involve tracking users without consent [12, 33]. In this scenario, fingerprinting is used to augment or replace cookies—e.g., to track across multiple domains, or when users disable or delete cookies. Our technique can be applied to either scenario.

Many fingerprinting techniques exist in the wild [20, 32, 57, 59]. Laperdrix et al. [48] published a survey that offers a detailed overview of existing fingerprinting techniques. They rely heavily on differences in devices' hardware and software characteristics found in HTTP header fields and JavaScript attributes. The key challenge is to identify features and attributes that further discriminate devices and allow for their unique identification, and to overcome the tendency of these features to evolve over time because of changes to the user's software, configuration, or environment.

2.2 GPU Programming

The *Graphics Processing Unit* (GPU) is specialized hardware for rendering graphics. GPUs have highly parallel architectures that are composed of multiple *Execution Units* (EUs), or *shader cores*, which can independently perform arithmetic and logic operations. Most consumer desktop and mobile processors from the past decade have on-chip GPUs with multiple EUs. For example, the UHD Graphics 630 GPU—integrated into Intel Core i5-8500 CPUs—includes 24 EUs.

Web Graphics Library (WebGL) is a cross-platform API for rendering 3D graphics in the browser [8]. WebGL is implemented in major browsers including Safari, Chrome, Edge, and Firefox. Derived from native OpenGL ES 2.0, a library designed for developing graphic applications in C++, WebGL implements a JavaScript API for rendering graphics in an HTML5 canvas element. WebGL takes a representation of 3D objects as a list of *vertices* in space and information

[1]The artifact accompanying this paper can be found at: https://github.com/drawnapart/drawnapart.

on how to render them, and translates them into a two-dimensional raster image that can be displayed on screen. WebGL abstracts this process as a pipeline. Two pipeline steps which are of interest to this work are the *vertex shader*, which places the vertices in the two-dimensional canvas, and the *fragment shader*, which determines the color and other properties of each fragment. The vertex and fragment shaders can run user-supplied programs, written in a C-derived programming language named *GL Shading Language* (GLSL).

2.3 Threat Model

We consider 3 different threat models for DrawnApart, corresponding to its applications in user tracking, login pages, and in defending against fingerprint spoofing.

Tracking Scenario. This is a destructive use case where a *tracker* employs browser fingerprinting to uniquely identify users across multiple browsing sessions, often to circumvent privacy protections such as cookie clearing or private browsing modes. In this scenario, the *tracker* aims to link distinct visits from the same device over time, creating a persistent identity. This allows the tracker to build a comprehensive profile of the user's behavior and preferences for targeted advertising or analytics, even if the user attempts to reset their online identity by clearing local state. We assume the tracker operates within the standard capabilities of a web application and does not have privileged access to the user's device. The user, in turn, does not actively spoof their hardware characteristics.

Login Scenario. This is a constructive use case where a *defender* website utilizes browser fingerprinting as a risk signal to authenticate returning devices and enhance user experience. In this scenario, a legitimate *user* attempts to log in to their account on the *defender* website. The *defender* collects a browser fingerprint during the login flow and compares it against a history of fingerprints. If the collected fingerprint matches a previously trusted device for this account, the *defender* may waive 2FA, treating the device as recognized. Conversely, if the fingerprint is novel or significantly different, the *defender* treats the login attempt as higher risk and requires stronger authentication to verify the user's identity. The goal is to reliably recognize the user's device despite natural variations in the fingerprint over time, distinguishing it from unknown devices or potential attackers.

Fingerprint Spoofing Scenario. This is a security-motivated use case where a *defender* website uses browser fingerprinting to detect spoofing attempts during authentication, as in the login scenario. In this scenario, an *attacker* aims to impersonate a legitimate *victim* by replaying the victim's deterministic fingerprint attributes to bypass risk-based 2FA challenges. In the spoofing scenario studied in this paper, an attacker aims to authenticate as a legitimate *victim* by presenting both the victim's stolen credentials and a spoofed fingerprint that makes the attacker's browser appear identical to the victim's previously observed device. Following prior work on fingerprint replay attacks [52], we assume the attacker has previously obtained a deterministic browser fingerprint collected from the victim (for example, via phishing or credential theft that is accompanied by fingerprint collection). The attacker uses a custom browser (or an extension) that can override JavaScript-exposed and protocol-level attributes, allowing them to replay the victim's deterministic fingerprint to the defender. In particular, the spoofed fingerprint includes the WebGL *renderer string*, which is commonly used to identify the victim's GPU stack. This threat model focuses on *impersonation*, not on a user who wishes to hide or randomize their fingerprint. In addition to the deterministic fingerprint, the defender collects a *non-deterministic* DrawnApart trace during the authentication flow. The defender's goal is to verify whether the observed DrawnApart trace is consistent with the claimed deterministic fingerprint. If the two are consistent, the defender may treat the device as recognized and waive 2FA; otherwise, the defender can require 2FA or flag the attempt as suspicious. A crucial assumption is that *DrawnApart traces are challenging to spoof.* A DrawnApart trace is obtained by measuring how the GPU responds to an arbitrary sequence of WebGL commands, which the defender can vary

across sessions. As a result, the attacker cannot anticipate the exact DrawnApart challenge ahead of time and cannot reliably pre-record a matching response for the victim device. Instead, the attacker is forced to submit a trace generated on locally available hardware (with a GPU stack that may differ from the victim's), enabling the defender to detect inconsistencies between the spoofed deterministic fingerprint and the collected non-deterministic signal. In this work, we focus on spoofing detection under the assumption that the challenge is unknown to the attacker prior to interaction, and we leave a systematic evaluation of the size and robustness of the challenge space to future work. To summarize, we assume the following about the attacker:

- **Objective:** Masquerade as the victim to gain account access.
- **Credential and fingerprint acquisition:** Possesses the victim's credentials and a previously collected deterministic fingerprint.
- **Deterministic spoofing capability:** Can replay deterministic fingerprint attributes (including the WebGL renderer string) via a custom browser or extension.
- **Limited foreknowledge:** Does not know the (potentially dynamic) DrawnApart challenge prior to the authentication interaction.
- **Non-deterministic spoofing limitation:** Cannot generate, on demand, a DrawnApart trace that matches the victim device, and therefore submits a trace collected on the attacker's own device.

We evaluate the tracking and login scenarios in Section 6 and the fingerprint spoofing scenario in Section 7.

3 GPU FINGERPRINTING

3.1 Motivation

Similar to past work [32, 49], we aim to uniquely identify devices. However, unlike previous work, which rely on the diversity of hardware and software configurations, we focus on distinguishing identical devices, which can significantly enhance the existing tracking capabilities. To do so, we incorporate techniques similar to the arbiter-based *Physically Unclonable Function* (PUF) concept of Lee et al. [50]. In an arbiter PUF, the statistical delay variations of wires and transistors across multiple instances of the same integrated circuit design are used to uniquely identify individual instances of the integrated circuit. In our case, we harness the statistical speed variations of individual EUs in the GPU to uniquely identify a complete system.

3.2 Design and Implementation

Access to the GPU is limited to the JavaScript, WebGL and WebGPU APIs, which present a high-level abstraction that challenges accurate timing and targeting of specific Execution Units (EUs) for fingerprinting. We overcome this by utilizing short GLSL vertex shader programs (Section 2.2) and exploiting the **non-randomized job allocation** within the WebGL stack, which predictably assigns parallel vertex shader tasks to different EUs. This allows us to issue commands that repeatedly target the same EU. Furthermore, we ensure the execution time of the targeted EU **dominates** the overall pipeline time by assigning non-targeted EUs a fast program while the targeted EU executes a computationally sensitive task.

Our technique generates a timing-based fingerprint (a *trace*) through the execution of a sequence of drawing operations, as illustrated in Figure 1. The full source code for these settings can be found in our artifact repository, as listed in Section 10. The implementation proceeds in three steps:

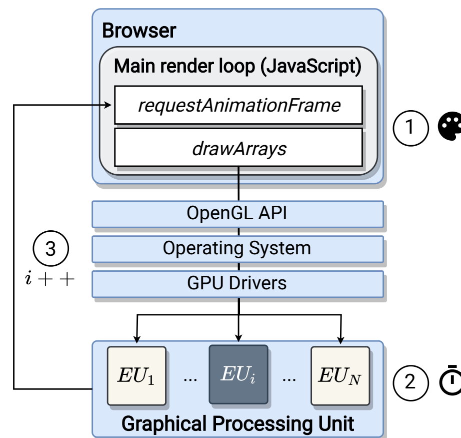

Fig. 1. Overview of our GPU fingerprinting technique: (1) points are rendered in parallel using several EUs; (2) the EU drawing point i executes a stall function (dark), while other EUs return a hard-coded value (light); (3) the execution time of each iteration is bounded by the slowest EU.

1. Render. We instruct the WebGL API to draw multiple points in parallel using the `drawArrays()` function. Points, being the simplest object to render, minimize pipeline noise. We invoke `drawArrays()` repeatedly, saving the execution time for each rendering process. We employ three measurement methods (**onscreen**, **offscreen**, and **GPU**, detailed in Section 5.1), which vary in computational load. The position of each point is determined by an attacker-controlled vertex shader.

2. Stall. The stall mechanism ensures that the rendering time is governed by the targeted EU. A single call to `drawArrays()` generates drawing operations differentiated by the hardware-generated vertex index, `gl_VertexID`. We created a GLSL vertex shader that examines this identifier and executes a computationally intensive *stall function* only if it matches the input variable `shader_stalled_point_id` provided by the JavaScript code.

In the **onscreen** setting, the vertex shader checks if `shader_stalled_point_id` equals `gl_VertexID`. In the **offscreen** and **GPU** settings, the shader uses a bit mask to check if bit `1 << gl_VertexID` is set. If the condition is met, the stall function executes (line 5 in listing 1); otherwise, the shader exits quickly, minimizing its contribution to the overall execution time.

```
1  uniform int shader_stalled_point_id;
2  void main(void) {
3    // Stall on this vertex?
4    if(shader_stalled_point_id == gl_VertexID) {
5      gl_Position = vec4(stall_func(),0, 1,1);
6    } else {
7      gl_Position = vec4(0, 0, 1 ,1);
8    }
9    gl_PointSize = 1.0;
10 }
```

Listing 1. Vertex shader with stall function, onscreen setting (GLSL).

3. Trace Generation. We execute the parallel drawing operation multiple times, changing the value of `shader_stalled_point_id` in each iteration to systematically iterate over and measure the relative performance of the different

EUs. The resulting output vector, a *trace*, contains the sequence of timing measurements corresponding to the time taken by the targeted EU to draw the scene.

We note that prior browser fingerprinting techniques extract **deterministic** fingerprints, which remain identical as long as the device's software and configuration have not changed. Our technique, in contrast, is based on timing measurements and, as such, is **non-deterministic**—multiple measurements made on the same device will return different values due to the effects of measurement noise, quantization, and the impact of other tasks running at the same time.

4 EVALUATION OVERVIEW

4.1 Motivation

We claim that our new method provides a tangible advantage over deterministic GPU-based fingerprinting. To establish this claim, we evaluate our system in a lab setting and in the wild.

In the **lab setting**, we assume the attacker can collect training traces from a set of identical machines (same hardware and software), running under identical environmental conditions. Next, the attacker is given a single trace and is tasked with identifying the machine that generated the trace. Our primary metric of evaluation in this setting is the **accuracy gain**, which measures the multiplicative gain in accuracy of a classifier that incorporates our non-deterministic method, when compared to a classifier which only uses deterministic inputs. An accuracy gain of 1 means that the classifier provides no advantage over traditional methods, while higher values show that it gives the attacker an advantage. The lab setting provides the most advantageous conditions for our classifier, for several reasons. First, existing deterministic schemes cannot tell apart identical devices, as we demonstrate experimentally, resulting in a very low base rate. Moreover, the attacker controls the setup including the attack parameters for each set of devices, the workload, the runtime environment and the collection duration.

We also evaluate our system **in the wild**. More specifically, we evaluate how our method can be applied to track devices from a set of over 2,500 machines with 1,605 distinct GPU configurations, recruited through a crowd-sourcing experiment. We first perform a standalone evaluation of our method, in the absence of additional identifying features. We then provide additional deterministic features to the classifier as listed on Listing 2. State-of-the-art fingerprinting techniques can produce unique browser fingerprints through the consideration of these signals, but these fingerprints are not ideal for tracking users since they evolve over time [74]. We therefore measure the added distinguishing power our method provides to existing browser fingerprinting schemes, with the primary metric of evaluation being the **additional tracking time** made possible through the combination of our novel technique with existing schemes.

The in-the-wild setting is more challenging. First, the technique must perform well across a large number of devices, precluding tailored attacks, and the attacker is prohibited from using any trace collection method that is overly intrusive or time-consuming. Second, the attacker's choice of machine learning pipelines is constrained. In particular, the attacker cannot use a long training phase since this does not make sense in the context of browser fingerprinting—the fingerprint should be useful at once, and not depend on the victim spending hours on the attacker's website. The attacker must also be able to accommodate new devices joining the dataset in real-time, and should not be required to spend multiple CPU hours retraining the classifier every time a new device is detected. Finally, the attacker cannot control the runtime characteristics of the machine being fingerprinted. Our method will have to be tolerant to workload variations, GPU payloads from other tabs, browser and system restarts, and so on.

```
1    cookies and session support,
2    HTTP headers: [Accept, Accept-Encoding, Language, User-Agent],
3    navigator: [DNT, platform, plugins],
4    screen: [width, height]
5    timezone,
6    WebGL: [vendor, renderer]
```

Listing 2. Deterministic attributes collected for the in-the-wild dataset.

In the following section, we study the lab setting to demonstrate an upper bound on our classifier's potential accuracy gain, and to investigate parameter choices and their trade-offs on accuracy, compatibility and performance. In Section 6, we select a single set of parameters and launch a large-scale crowd-sourced experiment in the wild, showing the advantage of our method in a realistic setting.

4.2 Machine Learning Pipelines

We use two machine learning approaches to evaluate our fingerprinting technique. In the **lab setting**, we cast our fingerprinting problem as a conventional multinomial classification task, where the input is the trace of N rendering times, and the output is the label of the device assumed to have generated this trace. We ultimately chose to use the Random Forest ensemble classification algorithm [22, 51] in the lab setting, as it empirically delivered the best classification results in terms of accuracy. We did not apply any feature engineering, submitting the raw traces into the classification algorithm. To make sure we did not overfit our model, we applied a 5-fold train-test split to the data, and collected the mean accuracy reported by the folds, as well as the standard deviation among folds.

To evaluate our system **in the wild**, we needed a more elaborate pipeline for the reasons listed in Section 4.1. Our method relies on *neural networks* and consists of several steps: (1) We preprocessed our traces by normalizing and reshaping them into matrix form. (2) We trained a convolutional neural network (*CNN*) to solve the multinomial classification task. (3) We transformed the classification network into an embedding network using the semi-hard triplet loss algorithm of Schroff et al. [67]. The resulting network is capable of transforming our trace into a representation called an *embedding*. Because of the way the network is designed, the Euclidean distance between two traces from the same device will be small, while the Euclidean distance between traces from different devices will be large. This allows the inference part of the classification to use the k-Nearest Neighbors classifier—given an unknown trace, measure the distance between its embedding and the embeddings of all known traces, and output the label of the embedding at the shortest distance. The simplicity of this classifier means the adversary can add new devices to the dataset simply by recording a few new traces and without retraining the entire network, a desirable property known as *few-shot learning*.

To ensure we did not overfit our in-the-wild model, we split our training dataset into two mutually exclusive parts, each with different labels, performed the evaluation on each part in isolation, and observed that the accuracies for each split were roughly the same. More details about the training process and dataset splits can be found in Section 6.

5 LAB SETTING

The objective of the lab setting is to discover DrawnApart's highest accuracy, and assumes that the attacker customizes the attack to the class of device and ignores aspects of detectability, compatibility or performance.

Evaluated Devices. Table 1 lists the devices used in the lab setting. We used 88 devices from nine distinct hardware classes, including desktops and mobile devices. The desktops include multiple generations of Intel processors, all running Windows 10, as well as a set of Apple Mac mini devices with Apple M1 chips, running MacOS X Version 11.1. Other

Table 1. Accuracy gains achieved under lab conditions

Device Type	GPU	Device Count	Timer	Base Rate (%)	Accuracy (%)	Gain
Intel i5-3470 (Gen 3 Ivy Bridge)	Intel HD Graphics 2500	10	Onscreen	10.0	93.0±0.3	9.3
			Offscreen	10.0	36.3±1.6	3.6
Intel i5-4590 (Gen 4 Haswell)	Intel HD Graphics 4600	23	Onscreen	4.3	32.7±0.3	7.6
			Offscreen	4.3	63.7±0.6	14.7
			GPU	4.3	15.2±0.5	3.5
Intel i5-8500 (Gen 8 Coffee Lake)	Intel UHD Graphics 630	15	Onscreen	6.7	42.2±0.7	6.3
			Offscreen	6.7	55.5±0.8	8.3
			GPU	6.7	53.5±0.8	8.0
Intel i5-10500 (Gen 10 Comet Lake)	Nvidia GTX1650	10	Offscreen	10.0	70.0±0.5	7.0
			GPU	10.0	95.8±0.9	9.6
Apple Mac mini M1	Apple M1	4	Offscreen	25.0	46.9±0.4	1.9
			GPU	25.0	73.1±0.7	2.9
Samsung Galaxy S8/S8+	Mali-G71 MP20	6	Onscreen	16.7	36.7±2.7	2.2
Samsung Galaxy S9/S9+	Mali-G72 MP18	6	Onscreen	16.7	54.3±5.5	3.3
Samsung Galaxy S10e/S10/S10+	Mali-G76 MP12	8	Onscreen	12.5	54.1±1.5	4.3
Samsung Galaxy S20/S20 Ultra	Mali-G77 MP11	6	Onscreen	16.7	92.7±1.8	5.6

than the Gen 10 devices, which had discrete Nvidia GTX1650 GPUs, all desktops used integrated graphics. For each class, the devices were purchased through the same order, configured with our University's official operating system image, and located in the same temperature-controlled lab. The mobile devices include multiple generations of Samsung Galaxy devices, all sourced through the Samsung Remote Test Lab [7]. All the mobile devices were Android-based and featured Samsung Exynos CPUs and Mali GPUs.

Comparison With Prior Fingerprinting Techniques. We reproduced and tested several state-of-the-art web-based fingerprinting techniques, including UniqueMachine [25], Fingerprint JS [4], and Clock around the Clock [66]. Our findings indicate that these methods are unable to distinguish between identical devices, as summarized in Table 2. Further details are available in our conference version [46].

Table 2. Accuracy of prior web-based fingerprinting techniques across identical device sets. We use the best values for DrawnApart for each generation (see Table 1). BR is the base rate, which is the accuracy of a random guess. N is the number of devices in each set.

Prior Fingerprinting Technique	**Gen 3 Accuracy** *(N=10, BR=10.0%)*	**Gen 4 Accuracy** *(N=23, BR=4.3%)*	**Gen 8 Accuracy** *(N=15, BR=6.7%)*	**Gen 10 Accuracy** *(N=10, BR=10.0%)*
UniqueMachine [25]	10.0%	4.3%	6.7%	10.0%
Fingerprint JS (FPJS) [4]	20.0%	4.3%	20.0%	10.0%
Clock around the Clock [66]	N/A	4.3%	N/A	N/A
DrawnApart	93.0%	63.7%	55.5%	95.8%

5.1 Tuning the Trace Parameters

We search for the parameter settings that provide the optimal accuracy gain for the different hardware configurations. Information about the stall function function operator selection and the number of points to render can be found in our conference version [46].

Timing Measurement Method. Scene rendering is performed in the GPU context, which is asynchronous to the CPU context. Simply measuring the time it takes the CPU to execute the draw operation, for example by calling `performance.now()` immediately before and after the call, does not provide any usable insight about the GPU. We therefore considered three measurement methods that are capable of measuring the actual drawing time of the GPU.

In the **onscreen** method, we render the scene to a standard HTML canvas element and then call `Window.requestAnimationFrame`. This function is passed a callback function that is called after the rendering is complete. Timing information is then extracted from within the callback. The onscreen method is the most compatible of those we evaluated and can be made invisible to the users, but browsers do not call `requestAnimationFrame` at a rate higher than the browser's maximum frame rate, which is typically 60 Hz. Thus, using this method requires that each iteration of our rendering operation take at least 16 ms to provide us with useful information, which causes a noticeable slowdown for the user.

In the **offscreen** method we use a worker thread and render the scene to an `OffscreenCanvas` object. This does not affect the user's main context and does not slow down the user. After rendering the scene, we call the `convertToBlob` method of the `OffscreenCanvas`, causing it to execute all instructions currently in the WebGL pipeline, and ultimately return a binary object representing the image contained in the canvas. We measure the time it takes to execute this command. Since there is no frame rate limit in this setting, each iteration of the rendering operation can take less time, allowing us to use more iterations.

The **GPU** method is the third method we evaluate. It is a modification of the offscreen method that does not measure timing on the CPU side. Instead, the WebGL disjoint timer query method is used to directly measure the duration of a set of graphics commands on the GPU side. To perform this measurement, we call `beginQuery`, issue the drawing operations, and call `endQuery`. Using `getQueryParameter`, we retrieve the elapsed time on the GPU side. This disjoint timer query command was previously used for side-channel attacks by Frigo et al. in their work in IEEE S&P 2018 [36]. As a result, support for this timer was disabled in Chrome version 65. However, with the introduction of Site Isolation [11], it was deemed safe to be re-enabled in Chrome version 70 [55]. In contrast to CPU-side timers, whose resolutions have been severely reduced to a few microseconds with jitter to mitigate against transient execution attacks [62], the GPU-side timer offers microsecond resolution with no jitter even on the most modern versions of Chrome [10]. This GPU-based timer thus has the potential to be the most accurate and the least sensitive to activity on the CPU side. On the other hand, its accuracy varies dramatically between different GPU architectures, and it is not supported by the commonly used Google SwiftShader renderer.

5.2 Results

Table 1 summarizes the accuracy gains obtained in the lab setting using different timing methods. The mobile devices were evaluated using the onscreen method only due to limited access to those devices. Gen 3 and Gen 4 are not evaluated using the GPU timer method since their hardware does not support it. All devices within each hardware class were sampled the same amount of times. We observed that our Random Forest-based classifier approaches peak accuracy as the size of the training data set approaches 500 traces per label. As the table shows, our scheme delivered significant accuracy gains, well above the base rate, in all scenarios, both for desktop and mobile devices. The parameter choices, however, did affect the performance of our scheme.

Effect Of Stall Function. As expected, each of the operators we evaluated performed differently on the different hardware targets.

In the offscreen setting, the `sinh` operator was consistently the best performer for the Gen 4 and Gen 8 corpora, while `mul` was better than `sinh` for the Gen 10 corpora. We hypothesize that since the offscreen setting allowed us to trigger multiple execution units at the same time, and the amount of advanced math units that handle trigonometric operations is lower than the amount of EUs, the conflicts and race conditions that arise inside the GPU gave this operator additional discriminating power.

Effect Of Timing Measurement Method. As stated above, the offscreen method allowed us to execute more iterations than the onscreen method, allowing us to capture data about EU contention, as well as on the timing of individual EUs. We were also interested in comparing the relative performance of the offscreen method, which measured time on the CPU side, and the GPU method, which used disjoint timer queries to measure performance on the GPU side. We hypothesizes that the GPU method would be superior to the offscreen method, since the GPU-side timer has higher accuracy than the CPU-side timer, and is not affected by the timing jitter introduced by inter-process communications (IPC) between the GPU and the CPU. In practice, we discovered that this is not always the case. As shown in Table 1, the GPU timer is better than the CPU timer for the Intel Gen 10 and Apple M1 corpora, has equivalent accuracy to the CPU timer on the Gen 8 corpus, and is actually less accurate than the CPU timer on the Intel Gen 4 corpus. To make matters worse, the disjoint timer query WebGL extension is not supported on several popular WebGL stacks, most significantly the software-based Google SwiftShader. Thus, the GPU-based timer is not appropriate for use in a large-scale experiment where the hardware configuration is not known beforehand.

5.3 Summary

Our results show that DrawnApart can tell apart identical computers in a controlled lab setting. Our next objective was to evaluate a realistic setting, in which the attacker has less control over the devices to be fingerprinted. We did so by first evaluating DrawnApart in a standalone setting, and then integrating it with a state-of-the-art browser fingerprinting algorithm.

6 IN-THE-WILD SETTING

Performing browser fingerprinting in the wild presents different challenges compared to what we experienced with the lab setting: (1) The lab evaluation assumed a closed list of devices. In the real world, new devices can be added at any time during the collection period, but we cannot re-train the model whenever it happens. (2) The lab evaluation assumed we had a long time to collect data and train over the devices. In the real world, we do not have unlimited access to a device so the collection of data must be fast. (3) Finally, the lab evaluation assumed the devices were idle and in a controlled environment. In the real world, we have to contend with variable computing loads, restarts, and updates to both the browser and the operating system. In order to understand the potential impact of DrawnApart in the real world, we collected 370, 392 traces from 2, 550 devices over 7 months and performed the two following evaluations:

- **Standalone evaluation**: Considering only DrawnApart traces without any other information, we aim to see how our method performs at re-identifying a device among others. In Section 6.2, we propose a one-shot learning pipeline whose aim is to match a new trace with another known trace present in our dataset.
- **Tracking over time**: Browser fingerprints evolve [32]. Vastel et al. developed two algorithms to track evolutions and link fingerprints that belong to the same device [73]. In Section 6.3, we show how DrawnApart can improve the FP-Stalker algorithms, which are the current state-of-the-art tracking algorithms, by increasing the duration users can be tracked. Our main metric to evaluate the gain of our technique will be the **median tracking time**.

Contrary to the standalone evaluation, we use all the attributes listed in Listing 2 as well as the DrawnApart traces.

6.1 Dataset constitution

Large-scale Experiment. To show DrawnApart's practical advantages over traditional deterministic fingerprinting methods as used in FP-Stalker, we launched a large-scale experiment with diverse hardware and software. We integrated our DrawnApart technique into the Chrome browser extension from the AmIUnique crowd-source experiment [49][2]. The extension periodically collects the browser fingerprints of thousands of volunteers, allowing us to track their evolution.

DrawnApart Collection Parameters. The crowd-sourced experiment constrained our choices. Most importantly, we wanted to be as non-intrusive as possible, as to not cause any user-perceivable slowdowns. In addition, we wanted to be compatible with various rendering stacks we encounter in the wild. Finally, we were interested in selecting a stall function that discriminates a wide variety of hardware. With these constraints, we selected the **offscreen** timing method, which allows fast measurements and is supported by all desktop versions of Chrome. We chose the `sinh` stall function operator, which provided good performance during our tests. We render all possible subsets of 10 points in each trace, for a total of $2^{10} = 1,024$ iterations per trace. Fingerprint collection takes a median time of 1.6 seconds. To increase our trace count, we repeated each collection seven times, for a median total run time of approximately 12 seconds. Since our offscreen collection script runs in a background worker thread, it does not affect the browser's interactiveness. Thus, our parameter choice creates a method that is non-intrusive and does not affect the user experience. It is, however, only effective if the user keeps the browser open for a sufficient amount of time. This makes it less suitable for websites that are loaded for very short periods of time, for example websites that take part in a chain of server-side redirects which is often found as part of the real-time bidding or attribution components of the online advertising ecosystem. We collected the traces every four hours.

Dataset Preparation. Our dataset contains 370,392 fingerprints from 2,550 unique devices. In each fingerprint, we collect the attributes listed in Listing 2, together with 7 DrawnApart traces. We identify devices with the same GPU by looking at the *WebGL renderer string* property. Over 90% of the devices shared a renderer string with at least one additional device. The largest observed group with same renderer string consisted of 534 unique devices.

We split our dataset into three subsets, divided by measurement time: 1MP contains 109,375 samples collected between 3-Jan-2021 and 7-Feb-2021, 2MP contains 46,293 samples collected between 7-Feb-2021 and 31-Mar-2021, and 3MP contains 214,724 samples collected from 3-May-2021 to 8-Jul-2021. We randomly choose 65% of the devices in 1MP that have more than 28 samples, and refer to this subset as $1MP_{65}$. The rest of 1MP will be referred to as $1MP_{rest}$ The limit of 28 samples, or 196 DrawnApart traces, was chosen to make sure the neural network will generalize well, by preventing it from overfitting on a small amount of traces of a specific device. We normalized each trace and reshaped a vector of length 1024 into a 32x32 matrix.

6.2 Standalone evaluation

Before integrating our model with FP-Stalker, we first evaluate it in isolation using only DrawnApart traces and ignoring the other attributes. In contrast to the classical ML model used in the lab setting, we used a neural network pipeline for the in-the-wild setting. The ultimate goal of the pipeline is to generate quality embeddings in Euclidean

[2]While the AmIUnique extension is also available for FireFox, the `OffScreenCanvas` feature we used for collecting traces was not supported in Firefox at the time the experiment was conducted.

space, which express the distance between traces. We begin the process by creating a *Convolutional Neural Network* (CNN)-based multinomial classifier. Details about the CNN architecture can be found in our conference version [46]. Our CNN achieved a training accuracy of 35.57% and a validation accuracy of 33.82%.

Semi-Hard Triplet Loss Model. The next step in our ML pipeline is the transformation of the multinomial classifier into an embedding, using the triplet loss method, which minimizes the distance between an anchor and a positive label and maximizes it against a negative label [67]. We took our trained classification model, removed its last layer, and retrained it for 30 epochs on the same dataset as before, with a bigger batch size of 1024 preprocessed traces and with semi-hard triplet loss. We took the weights of the epoch that yielded a model with the best accuracy using a 1-Nearest Neighbor classifier. The end-product of this process is a model that accepts preprocessed DrawnApart traces as input and produces embeddings in a Euclidean space. Labels are not involved in this process—we can take any DrawnApart trace, even from a device that the model was not trained on, feed it into the triplet loss model, and get Euclidean space embeddings.

Evaluating The Classifier. The use of embeddings mandates using a k-Nearest Neighbors classifier for analyzing the outputs of the network. Our metric for evaluation is the top-k accuracy, which stands for the probability that the correct answer is one of the k nearest neighbors of the selected trace, for $k = 1$, 5, and 10, according to the distance metric output by the model.

Base Rate Calculation. The accuracy of a classifier should be compared to the *base rate* obtained by a naive classifier with no access to the features. In the case of a classical learning problem, the naive classifier can observe the training data and learn the apriori probabilities of each label. Then, to get the best accuracy, this naive classifier will output the label of the most commonly observed device, or the n most commonly observed devices for a top-n setting. The base rate in that case is therefore the cumulative proportion of these devices in the dataset. In the case of a k-shot learning problem, the classifier does not know the apriori probabilities of each label, since it gets an equal amount of training data for each label. The naive classifier in this case will just output a random label, or n random labels for a top-n setting. The base rate in that case is only $n * (\#devices)^{-1}$.

Train-Test Split Evaluation. We evaluated our model in two ways: random train-test split, and k-shot learning. In the train-test split evaluation, we randomly split each of the $1MP_{65}$, $1MP_{rest}$ and 2MP datasets into two parts, using 80% for memorizing and 20% for testing. We first used $1MP_{65}$ for evaluation. To show that our network can generalize and work on traces it has never seen before, we next considered the performance of the network on $1MP_{rest}$. To show that our network generalizes to more devices and new traces, we evaluate it on 2MP. 2MP contains devices from 1MP, meaning that the neural network was trained on some of the devices in 2MP, but not all of them, but it was never trained on any traces from 2MP. The results in Table 3 demonstrate that our model accuracies are significantly better than the base rate for all of the three datasets. The accuracies on $1MP_{65}$ and $1MP_{rest}$ datasets are roughly the same, showing the model responds well to new devices. The small drop in the accuracy of 2MP despite a base rate of approximately half the other datasets, the addition of more devices and new traces and being collected at a later date, shows the model has generalized well.

k-shot Learning Evaluation. The k-shot learning evaluation was performed on the 2MP dataset. We chose 2MP to evaluate k-shot learning because we used the traces from $1MP_{65}$ to train our triplet loss model, which would bias the results. While some devices in this subset also appear in 1MP, none of the traces in 2MP were used to train or validate the neural network. In the memorizing phase, we memorize the first k collections ($k \times 7$ DrawnApart traces) of each device in 2MP. The rest of the traces of 2MP are used in the testing phase, again using a k-Nearest Neighbors classifier. This is an evaluation that is close to real-world use. An attacker would like to identify users with as few

Table 3. Standalone Performance of DrawnApart In the Wild Using The Random Split (RS) and k-shot Methods

Evaluation Method (Dataset)	Accuracy (*Base rate*) **Top-1**	**Top-5**	**Top-10**
RS ($1MP_{65}$)	28.88% (*1.00%*)	56.36% (*3.51%*)	68.70% (*6.15%*)
RS ($1MP_{rest}$)	28.28% (*1.22%*)	55.09% (*4.42%*)	67.15% (*7.20%*)
RS (2MP)	23.33% (*0.64%*)	47.23% (*2.78%*)	58.83% (*4.38%*)
1-Shot (2MP)	5.44% (*0.05%*)	14.10% (*0.26%*)	19.95% (*0.51%*)
5-Shot (2MP)	7.11% (*0.05%*)	19.34% (*0.26%*)	26.75% (*0.51%*)
10-Shot (2MP)	9.22% (*0.05%*)	22.77% (*0.26%*)	31.09% (*0.51%*)

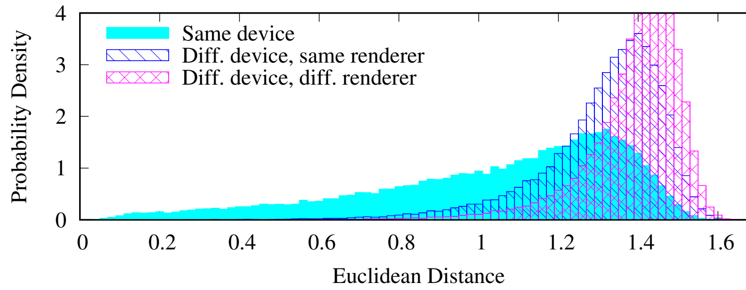

Fig. 2. Performance of the DrawnApart embedding function. A Euclidean distance below 0.65 indicates that the traces are likely to be from the same device.

collections as possible. This evaluation is harder than the previous one due to the small amount of data available for the memorizing phase. In addition, the time difference between 1MP and 2MP requires the network to deal with concept drift. As mentioned above, the base rate in this setting is very small, because the attacker cannot learn anything about the distribution of the devices in the test set. The results can be found in Table 3. As expected, they show a decrease in accuracy compared to the evaluation using random split, but our model still delivers significant accuracy beyond the base rate. We thus conclude that DrawnApart can be used for few-shot learning.

We leave the 3MP dataset to be used in the evaluation process of FP-Stalker to test the model on a truly unseen dataset that reproduces in-the-wild conditions.

Visualizing Euclidean Distances. To visualize the performance of our few-shot learning pipeline, we computed the Euclidean distances between pairs randomly sampled from 2MP from the three following populations: Embeddings from the same device, embeddings from different devices that share the same renderer string, and finally embeddings from different devices with different renderer strings. To eliminate correlation, only the first trace from each collection is used. Figure 2 presents the probability density of the different distributions. As the figure shows, embeddings from the same device get a lower Euclidean distance. Of interest is that embeddings from different devices that share the same renderer string have a lower Euclidean distance compared to different devices that do not share the same renderer string. This confirms that DrawnApart indeed fingerprints the GPU stack. We can also observe that if two traces have a Euclidean distance of less than 0.65, we can be almost certain that both traces came from the same device. We use this property in the next section to improve FP-Stalker.

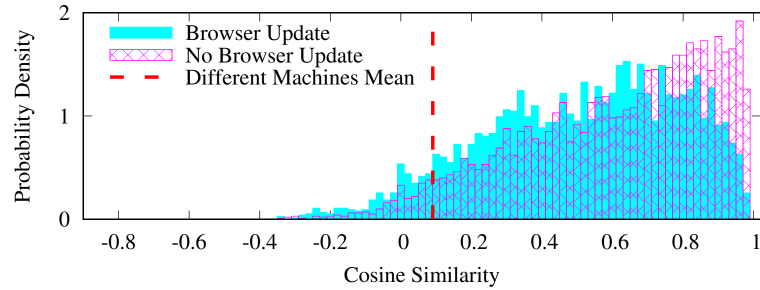

Fig. 3. Impact of browser updates on fingerprint consistency. The histograms show the cosine similarity between representative embeddings of consecutive visits, separated by whether a browser update occurred. The dashed red line marks the mean similarity between different devices (0.09), serving as a baseline for distinguishability.

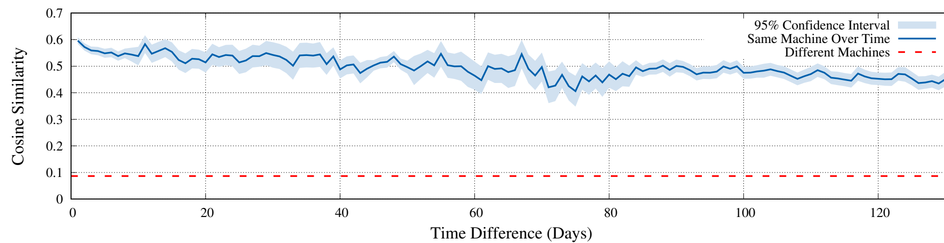

Fig. 4. Long-term stability of DrawnApart. The plot shows the average cosine similarity between the representative embedding of a device's first visit and its subsequent visits over time. The dashed red line indicates the average similarity between different devices (0.09).

Effect of Browser Updates. Browser updates can alter the rendering stack, potentially affecting DrawnApart's measurements. To assess this impact, we analyzed consecutive visits for each device in the 2MP and 3MP datasets. We categorized pairs of visits into two groups: those separated by a browser update and those without. For each visit, we computed a *representative embedding* by averaging the embeddings of its seven traces. We then measured the cosine similarity between the representative embeddings of consecutive visits. We use cosine similarity because it focuses on vector orientation (identity) and is robust to magnitude variations inherent in averaged L2-normalized vectors. Figure 3 displays the distribution of cosine similarities for both groups. The results show that browser updates have a measurable but limited impact. Although the average similarity decreases slightly when an update occurs, it remains significantly higher than the baseline similarity of 0.09 observed between different devices. This confirms that DrawnApart fingerprints retain sufficient consistency to identify devices even after browser updates.

Stability Over Time. To evaluate the long-term stability of our fingerprints, we measured the similarity between a device's initial representative embedding and those from subsequent visits in the 2MP and 3MP datasets. Figure 4 plots the average cosine similarity against the number of days elapsed since the first visit. The data reveals strong stability, with similarity remaining consistently high for over 130 days. A slight downward trend indicates a slow drift in embeddings over time. However, the average similarity for the same device remains far above the 0.09 baseline for different devices. This demonstrates that despite minor temporal drift, DrawnApart fingerprints remain distinct enough to distinguish a device from others over extended periods.

Summary. The results of the standalone evaluation, as summarized in Table 3, show a significant improvement over the base rate, demonstrating that DrawnApart is effective on its own. Furthermore, our evaluation confirms that DrawnApart remains stable over time and across browser updates. However, it can be observed that the classifier's effectiveness is significantly reduced in the k-shot model, where the attacker has a limited trace budget to be used for training. Putting these numbers into context is important. In the world of browser fingerprinting, no single attribute differentiates all devices. While some attributes are more discriminating than others, it is their combination that is key to differentiating one device from another. The standalone evaluation of DrawnApart shows that our method has the potential to significantly contribute to fingerprinting accuracy. In the following subsection, we empirically measure this contribution by using our method in conjunction with additional fingerprinting attributes.

6.3 Integrating DrawnApart with FP-Stalker

FP-Stalker is the state-of-the-art fingerprint linking algorithm [74]. In this section, we show that DrawnApart can be used to improve the state-of-the-art.

Hybrid Algorithm. FP-Stalker has two distinct algorithms: one entirely rule-based, while the other combines rule-based constraints and machine-learning. Vastel et al. demonstrated that their hybrid variant of yielded better results on their dataset, but was slower than its rule-based counterpart. As we are trying to prove the effectiveness of DrawnApart in a real-world scenario, we chose to implement and optimize the hybrid FP-Stalker algorithm, regardless of its speed.

FP-Stalker consists in: (1) a preprocessing step that discards fingerprints that contain inconsistencies, (2) a training phase, in which the Random Forest algorithm is trained on a balanced dataset, (3) an inference phase, in which the trained model, combined with rules, compares incoming fingerprints to a pool of previously classified fingerprints and attempts to link them. Algorithm 1 lists the linking algorithm.

Improving The Algorithm. Figure 2 shows that the Euclidean distance is efficient, to an extent, in differentiating devices. Based on the results obtained in Section 6.2, which show that DrawnApart can correctly classify devices with an acceptable accuracy, we decided to introduce the use of the generated embeddings as a complement to the machine-learning side of FP-Stalker. We note that the results of our nude FP-Stalker cannot be fully compared to the results obtained by Vastel et al. for two main reasons: (1) Their dataset spans for longer than the dataset we use in our experiments. (2) Flash-related attributes no longer exist [1], impacting FP-Stalker's effectiveness.

Integrating DrawnApart as a complement to FP-Stalker's machine-learning model is motivated by the fact that FP-Stalker uses a series of conditions on the output of the Random Forest that makes its decisions too restrictive. FP-Stalker's original code includes a function to optimize the threshold used by the Random Forest, which we adapted and ran on our dataset. The resulting threshold yielded similar results, consequently comforting our observation that the rules associated to the output of the Random Forest are too restrictive, and discard too many fingerprints coming from the same browser instance. On the other side, Figure 2 shows that even though the Euclidean distances can be used to efficiently differentiate devices with a relatively low threshold, its usage alone may yield an unacceptable rate of false linkages due to a little percentage of different devices having low Euclidean distances. To use DrawnApart embeddings in FP-Stalker, we average the seven embeddings that are collected with each fingerprint and we output an average embedding. We compute the cosine similarity between the previous and the new fingerprint's average embeddings. The resulting similarity is compared to a threshold we chose based on an analysis on the train dataset. This process is explained in the next paragraphs. If the similarity of the two embeddings is above the chosen threshold, we classify the

Algorithm 1: Hybrid matching algorithm with the DrawnApart addition highlighted in red

```
1  Function FingerprintMatching (F, f_u, λ, ϵ)
2      for f_k ∈ F do
3          if FingerPrintHasDifferences(f_k, f_u, rules) then  F_ksub ← exact ∪ < f_k > ;
4          else
5              exact ← exact ∪ <f_k>
6          end
7      end
8      if |exact| > 0 then
9          if SameIds(exact) then  return exact[0].id ;
10         else  return GenerateNewId() ;
11     end
12     for f_k ∈ F_ksub do
13         cosine_sim ← GetSimilarity(f_u.avg_embedding, f_k.avg_embedding);
14         if cosine_sim > ϵ then
15             return f_k.id
16         end
17         < x_1, x_2, .., x_m > = FeatureVector(f_u, f_k);
18         p ← P(f_u.id = f_k.id | < x_1, x_2, .., x_m >)
19         if p ≥ λ then
20             candidates ← candidates ∪ < f_k, p >
21         end
22     end
23     if |candidates| > 0 then
24         if |GetRankAndFilter(candidates)| > 0 then  return candidates[0].id ;
25     end
26     return GenerateNewId()
```

fingerprint as similar to the one being compared without further steps. The algorithm with the DrawnApart additions, is available in Algorithm 1.

Choosing The Epsilon Threshold. We chose the threshold by performing an analysis over similar and different devices on the train dataset. No data from the 3MP test set was used to tune this parameter. We generate an equally balanced dataset from the training set comprising the cosine similarity of similar devices and different devices, and compare different percentiles of the distance of each group. As opposed to the Euclidean distance used in Figure 2, we chose the cosine similarity for FP-Stalker because it is bounded by a more natural interval of $[-1; 1]$. Following our analysis, we noticed that the 5th-percentile of similar devices are all comprised below a similarity of 0.10. Consequently, we chose a threshold of 0.15 in our experiments to account for a safety margin.

Results. We executed our revisited FP-Stalker with its DrawnApart addition on the dataset described in Section 6.1. We first trained the Random Forest model on fingerprints in the 1MP subset. We then executed the lambda optimization in order to run FP-Stalker with its optimal parameters, as required by the original paper. Finally, we executed the inference phase on 3MP, which is unseen by the training phase of both FP-Stalker and the embedding's network. We execute both FP-Stalker without our contribution, and our revisited version with DrawnApart, on the same dataset for collection periods ranging from two to seven days. Table 4 presents the average tracking duration obtained for each collection period, with a top improvement of 66.66% compared to the original FP-Stalker on a collection period of six

Table 4. Average tracking time by collection period

Collection Period	Tracking duration in days		Improvement
	Nude FPS	FPS+DA	
2 days	17	26	+52.94%
3 days	17.25	25.5	+47.82%
4 days	17	28	+64.70%
5 days	17.5	27.5	+57.14%
6 days	18	30	+66.66%
7 days	17.5	28	+60.00%

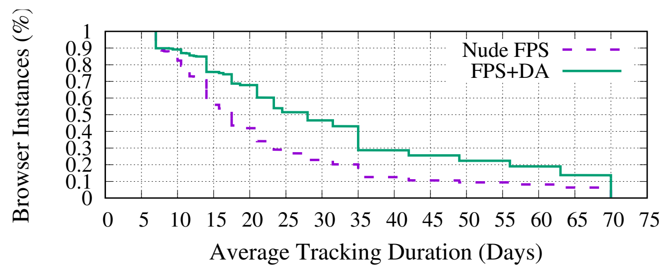

Fig. 5. Differences in Average Tracking Time between FP-Stalker (Nude FPS) and FP-Stalker with DrawnApart (FPS+DA)

days. Figure 5 presents the average tracking duration with a collection period of seven days, as presented in the original paper, which represents tracking a user who visits a website once a week. As the figure shows, adding DrawnApart to FP-Stalker increases the tracking time, raising the median average tracking time by **10.5 days**, from **17.5 days** to **28 days**. This is a substantial improvement to stateless tracking, obtained through the use of our new fingerprinting method, without making any changes to the permission model or runtime assumptions of the browser fingerprinting adversary. We believe it raises practical concerns about the privacy of users being subjected to fingerprinting.

7 TOWARDS SPOOFING DETECTION

Security-conscious websites often use enhanced security mechanisms to protect their users from unauthorized account access. One of the most commonly-used mechanisms is two-factor authentication (2FA). These enhanced security mechanisms, however, are known to disrupt the user experience, making them unpopular with users, despite their security benefits [78]. To minimize this disruption, websites apply risk-based heuristics to avoid the use of these invasive security mechanisms whenever possible. One such heuristic used to determine the risk of a login is the browser's fingerprint. As shown by Lin et al. [52], several high-profile websites, including banking and credit card company websites, suppress 2FA challenges when the user presents a consistent and recurring browser fingerprint. To abuse this heuristic, attackers can set up a phishing page that steals the user's credentials *together with the user's browser fingerprint*, and then forge the same browser fingerprint to masquerade as the victim's device. This is done by duplicating the target victim website's fingerprint collection code on the phishing website [65], and then using a specially-crafted browser extension on the attacker's device that injects the victim's fingerprint into the target website, instead of the attacker's true fingerprint. The increasing prevalence of fingerprint collection by phishing websites indicates that attackers may indeed be considering this strategy. As Sánchez-Rola et al. [65] note, 20% of phishing sites collect browser fingerprints.

More alarmingly, for 29.1% of login pages that utilize device fingerprinting, their corresponding phishing sites harvest the *entire set* of attributes that the benign login page collects, facilitating complete fingerprint spoofing.

The existence of browser fingerprint spoofing attacks raises questions about the use of browser fingerprints as a risk-based heuristic. In practice, website owners will have to choose between two non-optimal options: either waive 2FA for logins with the same fingerprint as a previously seen fingerprint, thereby exposing themselves to attackers who employ fingerprint spoofing, or otherwise require all users to use 2FA *for every login*, reducing usability and increasing user friction. In this section, we explore how DrawnApart can be used to effectively counteract fingerprint spoofing attacks. Our key observation, supported by Figure 2, is that DrawnApart traces can identify not only individual devices, but also the device's GPU hardware and software configuration, a value that is also represented in the deterministic fingerprint, in the form of the WebGL *renderer string*. Essentially, if the non-deterministic DrawnApart trace conflicts with the deterministic renderer string, it is likely that the entire fingerprint was spoofed. Since, as we claim below, DrawnApart traces are more difficult to record and play back than deterministic fingerprints, this provides a strong defense against fingerprint spoofing attacks.

7.1 Threat Model

We refer the reader to Section 2.3 for the full threat model. Briefly, we consider a phishing attacker who has obtained the victim's credentials and a previously collected deterministic fingerprint, and who uses a custom browser to replay this fingerprint (including the WebGL renderer string) to appear as the victim's device. During authentication, the defender additionally collects a non-deterministic DrawnApart trace and checks whether it is consistent with the claimed deterministic fingerprint.

Instantiation in our spoofing study. In Section 7, we instantiate this model by treating the renderer string as the attacker-controlled claim and the DrawnApart embedding as the observed signal: the attacker can perfectly replay the deterministic attributes, but—because the DrawnApart challenge is issued at interaction time—cannot reliably supply a trace that matches the victim's GPU stack on demand. Consequently, the attacker submits a trace generated on locally available hardware, and the defender flags the attempt when the resulting embedding is inconsistent with reference embeddings for the claimed renderer string (and, conversely, may waive 2FA when the two are consistent). This evaluation is intentionally scoped to the case where the attacker seeks to *impersonate* the victim, not to hide. That is, we assume the attacker replays a *coherent* deterministic fingerprint copied from the victim, so the login attempt has already passed simple inconsistency checks over attributes such as user agent, screen resolution, timezone, and renderer string and even FP-Stalker linking. Under this assumption, the relevant question is whether DrawnApart contributes an *additional*, non-deterministic signal where conventional deterministic methods fail.

7.2 Quantifying the Challenge Space and GPU Sensitivity

A critical assumption of our spoofing defense is the attacker's inability to supply a matching non-deterministic trace on demand. This is enforced by the vast challenge space available to the defender. A DrawnApart challenge consists of an arbitrary sequence of WebGL code, which serves as our stall function. At interaction time, the defender can dynamically select the mathematical stall function (e.g., sinh, mul, or a combination of multiple math functions). An experiment showing that different stall functions produce different classification accuracies, and thus different signals, is available in our conference version [46].

We denote L as the maximum number of WebGL commands in the stall function, M as the number of different mathematical WebGL commands available, and C as the total number of possible challenges. The challenge space is

calculated as:

$$C = \sum_{i=1}^{L} M^i = M^1 + M^2 + M^3 + \cdots + M^L$$

Consequently, the number of challenge permutations grows rapidly, preventing an attacker from pre-recording a comprehensive dictionary of responses. For instance, even a highly constrained setup where $L = 5$ and $M = 10$ allows the defender to generate $C = 111,110$ different challenges. Assuming a collection speed of one trace per second, an attacker would require over 30 hours of continuous rendering to pre-record this minimally-sized space. This is a very conservative estimate, as the defender can trivially increase L and M to expand the challenge space.

It is important to note that in this work we do not evaluate how different challenges affect DrawnApart's performance, and we leave a systematic evaluation of the challenges' performance to future work. For all experiments in this section, we use sinh as the stall function. This subsection is intended to provide a theoretical analysis of the size of the challenge space and its implications for spoofing defenses, rather than an empirical evaluation of specific challenges.

Furthermore, our evaluation inherently accounts for attackers attempting to spoof traces using an identical GPU model and renderer string. In our in-the-wild dataset, over 90% of devices shared a renderer string with at least one other device. Figure 2 demonstrates that while different devices sharing a renderer string produce embeddings that are closer together than entirely distinct GPUs, they remain distinguishable. This confirms that even highly resourced attackers utilizing identical hardware cannot reliably evade detection.

7.3 Methodology for Spoofing Detection

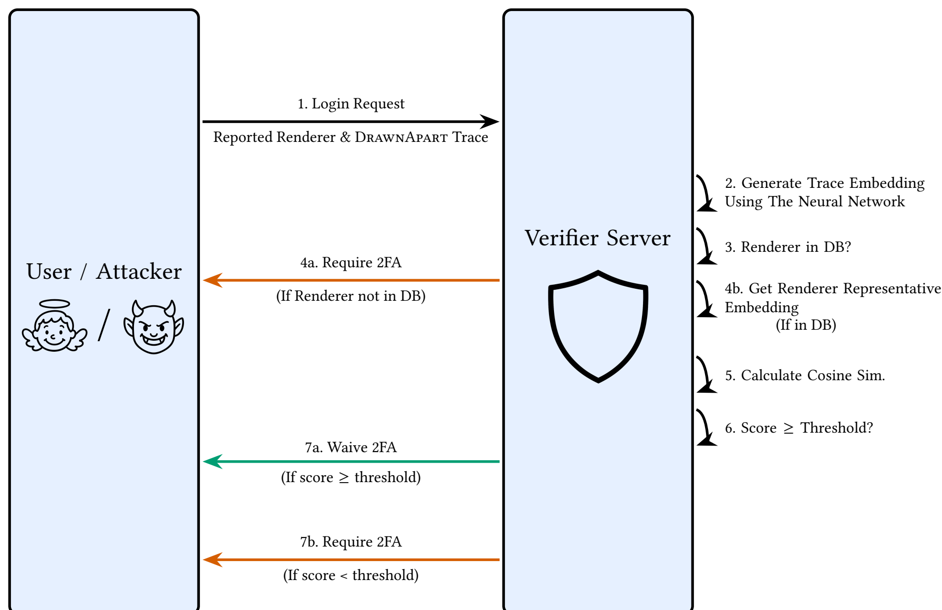

Fig. 6. Diagram of the DrawnApart spoofing detection inference pipeline.

Inference Pipeline. To employ DrawnApart for spoofing detection, the defender needs to verify if the reported renderer string matches the true renderer string of the user's device. The proposed inference pipeline is illustrated in

Figure 6. We view this pipeline as a *second-stage* check that complements existing deterministic defenses rather than replace them. In a realistic deployment, a defender would first apply standard consistency checks and rule-based filters over deterministic attributes. DrawnApart is then useful for login attempts that remain plausible after those checks because the attacker is replaying the victim's deterministic profile consistently. Through our proposed method, we first obtain the trace's embedding by applying a deep learning encoder to the DrawnApart trace. The model used is a deep neural network trained with the renderer string as its label, selected due to its superior performance in our evaluation (see Table 6). Further details on this model and other considered configurations are provided in the 'Neural Network Training' subsection, and they all utilize the training methodology described in Section 6.2. We compute the mean embedding for this specific renderer string by averaging the embeddings of all devices with a similar renderer string found in our database. We refer to this embedding as the *representative embedding*. Finally, we measure the cosine similarity between the trace's embedding and this representative embedding to obtain a similarity score. A similarity score above a predefined threshold indicates that the incoming trace as benign, and that 2FA can be waived for this login. If the renderer string reported by the user's browser is not in our database, we cannot determine if the incoming trace is spoofed. In such cases, the user will be required to use 2FA for login.

Representative Embeddings Database. We compute and store the mean embedding for each renderer string using the embeddings of traces in the training set. This means the current evaluation is limited to renderer strings present in the training set. However, since our model is capable of few-shot learning, new renderer strings can be added to the database without retraining the model. Evaluating the scenario of introducing new renderer strings to the database is left for future work.

Neural Network Training. We evaluated three different labeling techniques for generating embeddings, to find which is most effective at detecting spoofing attempts. The training dataset for all of these neural networks was $1MP_{65}$, as described in Section 6.

(1) The first network we considered was a neural network trained using semi-hard triplet loss with the **device ID** of each fingerprint as its label, similar to the one we trained and used in Section 6.2.
(2) The second neural network was trained using the same methodology as in Section 6.2, but used only the **renderer string** of each fingerprint as its label. Renderer strings "Missing" and "Not supported" were excluded from the training set to ensure the neural network learns from high-quality annotated data. This approach trains the neural network to create effective embeddings for DrawnApart traces based on their renderer strings.
(3) The third neural network was trained similarly, but using the **GPU model** extracted from the renderer string of each fingerprint as its label. We extracted the GPU models using ChatGPT 3.5, with manual correction of errors by two independent annotators. Some renderer strings, such as *ANGLE (VMware SVGA 3D Direct3D11 vs_5_0 ps_5_0)*, do not include a specific GPU model, and were consequently filtered from the training set. Analyzing this neural network's output helps determine whether aggregating renderer strings by GPU model enhances generalization, or if using the renderer string directly is preferable for training. We note that we did not parse the renderer string to extract the GPU model during evaluation, since the GPU model extraction method we described is not feasible for low-latency scenarios.

Table 5 provides an overview of the labeling techniques used. Given that multiple devices can share the same renderer string and multiple renderer strings can share the same GPU model, the device ID labelling technique has the highest number of distinct labels in the training set, while the GPU labeling technique has the least.

Table 5. Summary of labeling techniques used to train the neural networks. Each row represents a labeling technique used to train a different neural network.

Labeling Technique	Label Example	# of Distinct Labels In Training
Device ID	Device 42	714
WebGL Renderer String	(Intel Mesa Intel(R) UHD Graphics 620 (KBL GT2) OpenGL 4.6 core)	226
GPU Model	Intel(R) UHD Graphics 620	168

Metrics. We define a positive case as a spoofing attempt and a negative case as a benign visit. Under this definition, the true negative rate (TNR) is the proportion of benign visits correctly classified as such, for which 2FA is waived, and the true positive rate (TPR) is the proportion of spoofing attempts correctly identified, for which 2FA is required. Conversely, false positives are benign visits incorrectly flagged, requiring 2FA, while false negatives are spoofing attempts misclassified as benign, bypassing 2FA. The metric we chose for evaluating our spoofing detection scheme is the true negative rate at a given true positive rate, denoted as TNR@TPR. The choice of a confidence threshold involves a trade-off between TNR and TPR: a higher TPR (detecting more spoofing attempts) typically results in a lower TNR (waiving 2FA for fewer benign visits). Given the security-critical nature of this domain, prioritizing the detection of spoofing attempts (higher TPR) is paramount, even if this results in a lower TNR, i.e., fewer benign visits for which 2FA is waived. We thus aimed for a high TPR and evaluated its impact on the TNR. Specifically, we evaluated our method at two working points: TNR@TPR=0.99 (TNR at a TPR of 0.99) and TNR@TPR=0.999 (TNR at a TPR of 0.999).

7.4 Evaluation of Spoofing Detection

The evaluation was performed on the 2MP dataset, excluding traces with renderer strings “Missing” or “Not supported”, which together constitute 0.96% of the dataset. These traces were omitted because their renderer strings cannot be verified.

Figure 7 presents the distributions of cosine similarities between DrawnApart embeddings and representative embeddings of renderer strings. The figure shows two distributions: one for cosine similarity to the mean embedding of the trace’s actual renderer string (representing benign visits), and another for cosine similarity to mean embeddings of different renderer strings (representing spoofing attempts). We can observe that cosine similarities for benign visits are generally higher than those for spoofing attempts. Ideally, the two regions (*Benign visits* and *Spoofing attempts*) would not overlap, allowing us to extract a clear threshold. While our current method does not achieve this ideal separation, it still allows for selecting thresholds that correctly classify a non-negligible number of benign visits (benign visits that 2FA will be waived for, which we denoted as true negatives), while almost all the spoofing attempts are to its left (spoofing attempts that 2FA will be required for, which we denoted as true positives). In summary, this figure demonstrates that DrawnApart can detect nearly all spoofing attacks, while correctly identifying a considerable portion of benign visits.

The results for the spoofing detection evaluation are presented in Table 6. The best results were achieved when using the renderer string as the label for training the neural network. Surprisingly, using the GPU model as the training label yielded the poorest results. We hypothesize this is because the neural network focused on minimizing distances between embeddings with the same GPU model, an aggregation of multiple renderer strings. This aggregation likely made it harder to distinguish between different renderer strings sharing the same GPU model. Although we initially

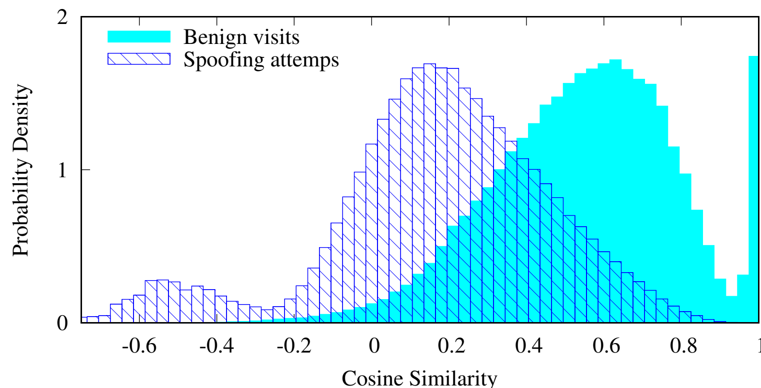

Fig. 7. Cosine similarity between the DrawnApart embeddings and the representative embeddings of the actual renderer string, labeled as "Benign visits" and renderer strings that are not the device's renderer string, labeled as "Spoofing attempts". The embeddings are generated using the neural network that was trained using the renderer string as the label.

believed this aggregation might improve generalization, it ultimately degraded performance. The results indicate that DrawnApart can waive 2FA for 19.4% and 9.0% of benign visits while detecting 99% and 99.9% of spoofing attempts, respectively. These outcomes are achieved by setting the cosine similarity thresholds at 0.752 and 0.857, respectively, as shown in Figure 7. In other words, any login attempt with a cosine similarity above the respective threshold will have 2FA waived. This initial result demonstrates a promising defensive application for hardware fingerprinting, and highlights an important research direction for increasing the defense's accuracy and effectiveness. Our method becomes more effective as the challenge space for creating the DrawnApart trace grows, making it more difficult to record and replay the DrawnApart trace, as discussed in the threat model.

Table 6. Results for spoofing detection.

Neural Network Training Labels	TNR@TPR=0.99	TNR@TPR=0.999
Device	0.140	0.026
Renderer String	**0.194**	**0.090**
GPU Model	0.111	0.024

7.5 Comparison With Previous Works

Picasso. Bursztein et al. proposed Picasso [23], which aims to identify inconsistencies between the operating system and browser reported by the user's user-agent and the operating system and browser found by HTML5 canvas fingerprinting. Picasso constructs a HTML5 canvas hash such that the hash is consistent across different devices with the same operating system and browser, but inconsistent across different operating systems and browsers. To mitigate replay attacks they introduce a challenge-response mechanism to their fingerprinting method by using the drawn elements as a challenge. At the time of its publishing, Picasso was able to identify different operating systems and browsers with 100% accuracy. There is, however, a much smaller diversity of browsers and operating systems than there exists for computer hardware. Thus, it would be practically possible for an attacker to have a device per common operating system with all the

common browsers installed on each device. This will give an attacker the ability to dynamically generate the correct Picasso response and bypass Picasso's authentication.

Morellian Analysis. Laperdrix et al. proposed Morellian Analysis [47], an HTML5 canvas fingerprinting-based authentication scheme resistant to trivial replay attacks. Their method involves drawing a canvas based on a challenge and storing the resulting canvas hash on the server. For subsequent visits, the device must draw the same canvas with the same challenge for authentication and draw a new canvas with a new challenge for the next visit. They evaluated their method for TPR, showing a TPR of 1 when the browser supports the canvas API. However, they did not provide an evaluation for TNR.

Both Picasso and Morellian Analysis utilize canvas fingerprinting to detect spoofing attempts. Our approach, however, is fundamentally different as it is non-deterministic. DrawnApart can similarly complement Morellian Analysis as it does FP-Stalker Section 6.3. In cases where Morellian Analysis fails to match the canvas hash with a previous visit, DrawnApart can determine whether it is a spoofing attempt or simply a change in the deterministic fingerprint.

7.6 Discussion

We investigated the effectiveness of DrawnApart in detecting spoofing attempts. Our method empowers website owners to tailor their risk tolerance by selecting the threshold for cosine similarity between incoming traces and the representative embedding of the renderer string. By setting the risk at either 1 in 100 (TPR of 0.99) or 1 in 1000 (TPR of 0.999) spoofing attempts being mistakenly classified as benign visits, DrawnApart enables 2FA authentication to be waived for approximately 1 in 5 users or 1 in 11 users, respectively, increasing usability.

Our current evaluation has several limitations. First, the method is restricted to known renderer strings. Users with renderer strings not in the database will be required to undergo 2FA. While new renderer strings can be added to the database without retraining the model, the method's effectiveness on these new strings remains future work. Second, our method reduces, but does not eliminate, the risk of spoofing, as it cannot detect all spoofing attempts due to the non-deterministic nature of DrawnApart. Third, the spoofing detection method will not function for benign users whose WebGL renderer string is "Missing" or "Not supported," as their renderer string cannot be verified. These users will require 2FA, which increases friction but mitigates spoofing attempts from such configurations. Fourth, DrawnApart currently does not employ a dynamic challenge for trace creation, meaning an attacker could potentially record and replay a DrawnApart trace, as discussed in the threat model. We leave the evaluation of the effectiveness of our method in preventing fingerprint playback to future work.

To summarize, this section presented a novel defensive application of DrawnApart for spoofing detection. Even at this initial stage, our method can be immediately deployed to reduce the frequency of 2FA challenges, thereby preserving usability and convenience while enhancing website security compared to the current practice of waiving 2FA based solely on deterministic fingerprint matches. We demonstrated that training the neural network using the renderer string as the label yields the best results for detecting spoofing attempts, compared to training with device ID or GPU model labels. The effectiveness of DrawnApart in detecting spoofing attempts underscores the value of exploring non-deterministic device behaviors for security applications. As the landscape of online threats evolves, such novel approaches to verifying device integrity become increasingly crucial. The ongoing development of robust, user-friendly authentication methods remains a critical area of focus for the security community. To make our method safe from replay attacks, we propose adding a dynamic element to the DrawnApart challenge, a promising direction which will be explored in the future.

8 DISCUSSION

8.1 Ethical and Privacy Implications

We integrated our fingerprinting algorithm into the Chrome browser extension from the AmIUnique crowd-sourced experiment in January 2021. On the installation page, users are informed of its purpose and of the data that is collected. To safeguard users' privacy, collected traces are only associated with a random identifier created when the extension is installed, and participants can delete all their data by submitting their extension ID. Out of an abundance of caution, we decided not to publish the weights of the triplet loss model trained on these users, since it can enable an attacker to track these users. The extension and the handling of collected data conform to the IRB recommendations we received.

We now discuss the deeper implications and potential for abuse of DrawnApart. DrawnApart shares the same ethical and privacy implications with existing browser fingerprinting techniques – namely, that a permanent identifier can now be attached to a user without their consent, and that this identifier can be used to track the user across websites and over time, without respecting their expressed preferences (e.g., via Do Not Track headers, Incognito mode or cookie blocking). Existing work on the topic of browser fingerprinting has discussed these implications in depth [19]. A question which is worth exploring is: does the technical novelty and increased accuracy of DrawnApart, compared to prior techniques, warrant a different treatment of these implications? We argue that the answer is negative, for two reasons. First, despite the different technical form in which DrawnApart collects its fingerprint, when compared to prior techniques, the end result, as it is reflected to the user, is the same: a unique identifier which can be used to track them. Second, despite the increased accuracy of DrawnApart, it still does not provide a perfect, permanent, and undeniable identifier for all online users. Such an identifier, indeed, would completely eliminate the possibility of anonymity online, which would have far-reaching implications for freedom of speech, political activism and other fundamental rights. Fortunately, DrawnApart does not reach this level of accuracy.

8.2 Fingerprinting Countermeasures

Countermeasures can be divided into three groups.

Blocking Scripts. Filter lists block resources known to be a threat to user privacy. This is the case of Brave's Shield mechanism [2] and extensions, such as Ghostery [5] or Privacy Badger [6]. However, filter lists against trackers and fingerprinting have been shown to lack exhaustiveness [35, 41].

API Blocking. Tor Browser, by default, and Firefox, with specific configuration, prevent web pages from reading out the contents of the canvas for privacy reasons. Our technique does not examine the canvas content, but rather measures the time required to draw different graphics primitives. Snyder et al. [70] consider the WebGL specification a "low-benefit, high-cost standard", which is required by less than 1% of the Alexa Top 10k websites. This may lead some people to consider the extreme option of completely blocking WebGL, as possible way of preventing GPU fingerprinting. Disabling WebGL, however, would have a non-negligible usability cost, especially considering that many major websites rely on it, including Google Maps, Microsoft Office Online, Amazon and IKEA. As a form of compromise, we note that Tor Browser currently runs WebGL in a "minimum capability mode", which allows some WebGL functionality while preventing access to the `ANGLE_instanced_arrays` API used by our attack.

Changing Attribute Values. Defenses can change an attribute value either to make it similar with common values shared by a large proportion of users, or to add noise to it. For example, Tor Browser unifies the values of many attributes for all users so that their fingerprint is identical, and some browser extensions add noise to rendered canvas images [73]. Wu et al. [79] introduced a countermeasure that eliminates the differences in floating point operations

during the rendering process to eliminate the differences in the rendering composition of WebGL. Blurring defenses on canvas and WebGL focus on changing values. Our technique does not directly rely on the differences in images in a rendering process, and therefore is not affected by the countermeasure of Wu et al. [79].

There are three elements that are crucial to our fingerprinting technique: the ability to issue drawing operations in parallel. The entire graphics stack tendency to deterministically choose which EU will render each vertex. And the ability to measure the time it takes to render. Disrupting any of these elements could affect the accuracy of our technique.

Preventing Parallel Execution. To block our method, graphics stack could limit each web page to a single EU, or disable hardware-accelerated rendering altogether and use a deterministic software-only pipeline [79]. However, this would severely affect usability and responsiveness, because WebGL is built around massive parallelism. Existing graphics APIs do not also support partitioning execution to a subset of EUs at the moment.

Preventing Deterministic Dispatching. Adding a randomization step to the GPU's dispatcher would make it impossible for the web page to choose which EU receives which vertex. Assuming the dispatcher still attempts to fill up all available EUs, the effect on performance can be minimized. We note that this countermeasure is not perfect, since a permuted trace still contains data about the system being fingerprinted.

Timer Degradation (a.k.a. jitter). To degrade timer precision and mitigate high-precision timing attacks like Spectre [44], browsers such as Chrome and Firefox have introduced random uniform noise [61, 72], commonly referred to as jitter, into time measurements. This countermeasure was adopted after Schwarz et al. [68] demonstrated that attackers can gain a high-precision timer using clock interpolation. For example, Chrome currently adds a 100-microsecond jitter [54]. Our standard experiments were performed with jitter enabled, at its default value, demonstrating our method's robustness to current levels. To assess the impact of stronger defenses, we simulated increased jitter by adding random noise from a uniform distribution $[0, \text{jitter_max_magnitude}]$ to our traces. Figure 8 depicts the accuracy on the Gen 4 Offscreen dataset as jitter increases. Although accuracy decreases from 63.5% at the default 100-microsecond level to 37.6% at 300 microseconds and 31.3% at 1100 microseconds, it remains well above the 4.3% base rate (random guessing among 23 devices). This suggests that while substantial increases in jitter (e.g., to 300 microseconds) reduce performance, our technique retains effectiveness even under highly degraded timing conditions.

Profiling the execution of our scripts. DrawnApart's trace collection exercises the GPU stack in a structured way: it repeatedly dispatches near-identical GPU workloads while sweeping a control parameter (e.g., the "stalled" work-item/vertex) and synchronizing to obtain per-iteration completion times. This produces a distinctive interaction pattern at the JavaScript - graphics boundary: many back-to-back submissions with minimal semantic dependence on rendered output, combined with systematic timing collection. Since current JavaScript engines already perform runtime profiling to drive optimization decisions, a browser could leverage this existing instrumentation to detect "profiling-like" scripts that exhibit: (1) repeated GPU dispatch loops with little or no user-visible effect (2) fine-grained synchronization/timing after each dispatch (3) and parameter sweeps consistent with isolating microarchitectural resources Such a detector could then trigger a blocking defense to disable or disrupt one of the enabling conditions for DrawnApart—parallel execution, deterministic dispatch, and time measurement.

Preventing Time Measurements. Countermeasures that reduce, or even disable, the availability of timer APIs can affect our technique, but completely blocking timing measurements from the web is known to be a futile task [68, 69].

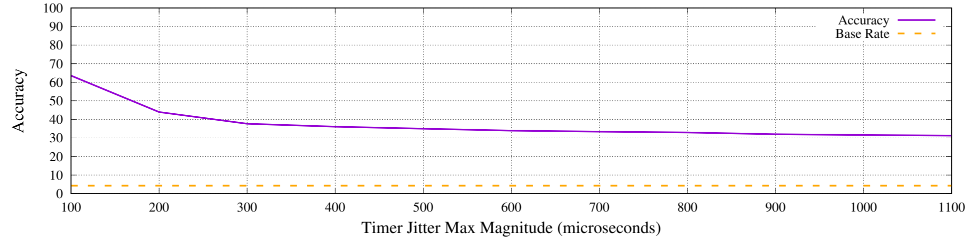

Fig. 8. Accuracy as a function of the jitter added to time measurements on Gen 4 devices. The base rate is 4.3%, highlighted with a dashed orange line.

8.3 Limitations and Insights

Temperature and voltage control limitations. The in-the-wild, crowd-sourced experiments demonstrate that DrawnApart can work successfully in conditions that are not under the attacker's control. In contrast, our lab experiments cover a limited set of conditions. Specifically, we showed that temperatures between 26.4 °C and 37 °C had no impact on the results. However, we cannot preclude the possibility that temperatures outside this range do not affect the results. Similarly, our lab experiments did not control for GPU voltage variations, which could affect our fingerprinting capability. These limitations notwithstanding, the results of the crowd-sourced experiments do provide confidence that DrawnApart is effective at scale in diverse operating conditions.

Device restarts. To evaluate the effect of device restarts on fingerprinting accuracy, we trained a model on the Gen 3 devices and tested it against post-reboot traces, obtaining an overall accuracy of 50.3%. We observed that the drop in accuracy is non-uniform; some devices maintain stable fingerprints across restarts, whereas others change significantly. While we do not track reboots in our in-the-wild experiments, the duration of the study ensures that our results inherently account for any reboot-related accuracy drops.

Operating System Reinstalls. We did not test OS reinstalls in our lab setting. And we are unable to identify reinstalled operating systems in our in-the-wild corpus due to the manner the data is collected (i.e., through our browser extension). Their impact on the accuracy of DrawnApart remains unevaluated.

Browser versions. We evaluate our technique across ten Chrome versions ranging from 80.0.3987.116 to 81.0.4044.138. These span two major releases: six minor versions in v80 and four in v81. Training our classifier exclusively on the latest v80 version (80.0.3987.163) achieved ~90% accuracy across all v80 versions, but dropped significantly to ~60% when tested against v81. Conversely, a combined model trained on traces from both major versions maintained a consistently high accuracy of 90±% across the entire evaluation set. The Chromium changelog shows over 10,000 commits between these versions, including several hundred affecting the GPU and WebGL API [9]. We hypothesize that these underlying WebGL stack modifications prevent a v80-only classifier from generalizing to v81. This highlights the importance of training on diverse browser versions to obtain a robust fingerprinting model. We explore the impact of browser updates more deeply in our in-the-wild experiments (Section 6).

Scalability. While our in-the-wild evaluation spans 2,550 devices, we anticipate DrawnApart to scale effectively to populations of tens or hundreds of thousands of devices. DrawnApart is designed to operate as a signal within a broader fingerprinting pipeline. In a large-scale deployment, deterministic attributes (e.g., User Agent, screen resolution, WebGL renderer string) would first constrain the search space. Scaling to larger populations introduces two challenges: computational overhead and potential false positives. Computationally, finding the nearest neighbor in a high-dimensional

space is a well-optimized problem. Libraries like Faiss [30] can perform similarity searches over millions of embeddings in under a millisecond [29], ensuring negligible latency impact. Regarding accuracy, a larger population increases the probability of collisions (i.e., false positives). However, as detailed in our FP-Stalker improvement in Section 6, we employ a dynamic similarity threshold derived from the training set's statistics. This allows system operators to tune the trade-off between false positives and false negatives. By adjusting this threshold and relying on the initial filtering of deterministic attributes, we expect DrawnApart to maintain high tracking performance even as the population scales to hundreds of thousands of devices.

8.4 Future Work

In-depth Root Cause Analysis. We shared our work with a committee of WebGL experts in an effort to investigate the root cause of DrawnApart. They acknowledged that the results reported in the paper offer insight on the tracking implications that WebGL can introduce and that our method can highlight differences introduced by the hardware manufacturing process. They propose additional hypotheses for the mechanism through which manufacturing variations enable DrawnApart. Specifically, the two proposals are that: (1) DrawnApart might be uncovering differences in power consumption. A study by von Kistowski et al. [76] noticed differences in power consumption from identical CPUs under the same load but it remains to be seen if and how this could translate to GPUs and WebGL. (2) The effect might be induced by a difference in the response to temperature curves. Validating either hypothesis requires detailed knowledge of the design and the manufacturing process, which are only available to the manufacturers, and are likely beyond the scope of a typical academic research.
Next-Generation GPU APIs. DrawnApart currently only uses the WebGL API, limiting its speed and accuracy. Upcoming Web-based compute-specific GPU interfaces may allow for far more efficient fingerprinting. There are two compute-specific GPU APIs for web browsers: WebGL 2.0 Compute and WebGPU. WebGL 2.0 has been disabled in 2020 [64], and efforts were directed towards WebGPU [43]. WebGPU is currently under active development, and is supported in Chromium based browsers, such as Chrome, Edge, and Opera, but is not supported in the stable version of Safari and Firefox.

These APIs introduce *compute shaders*, a form of computational pipeline that coexists with the existing graphics pipeline. One significant feature offered to compute shaders is the ability to synchronize among different work units, by using atomic functions, message queueing or shared memory. We used this synchronization primitive to prototype a faster fingerprinting technique for WebGL 2.0 Compute. In our prototype, all workers race to acquire a mutex, and we record the order in which the different work units were granted the mutex. We tested this fingerprinting technique on our Gen 3 corpus, after enabling WebGL 2.0 Compute support in Chrome through a command-line parameter. This compute-based fingerprint delivered a near-perfect classification accuracy of 98%, while taking only 150 milliseconds to run, much faster than the onscreen fingerprint which took a median time of 8 seconds to collect. We believe that a similar method can also be found for the WebGPU API once it becomes generally available. The effects of accelerated compute APIs on user privacy should be considered before they are enabled globally.

9 RELATED WORK

Web-based Fingerprinting. Eckersley [32] was the first to show that it is possible to fingerprint browsers based on their features and configurations. Mowery et al. [56] classified web fingerprinting use as constructive or destructive. Constructive fingerprinting can detect bots [23, 42, 75], or help to protect sign-in processes [16, 47]. Conversely, destructive use can track users and their browsing habits. Many browser attributes are considered parts of a browser

fingerprint, including `navigator` and `screen` properties [32, 49], font enumeration [60], audio rendering [33], and the WebGL canvas [57]. These techniques are all unable to tell apart identical devices.

Mobile Fingerprinting. Mobile devices have less hardware and software diversity compared to desktops [37]. However, they possess additional fingerprinting sources such as sensors [21, 28, 80], microphones [26, 81] and cameras [18]. Manufacturing variations can also manifest as differences in the radio frequency (RF) behavior of networked devices [14, 15]. These techniques are tailored to mobile and RF environments, while our technique works in all browsers that support WebGL, without requiring permissions, additional sensors or RF hardware.

Physically Unclonable Functions. The silicon-based physically unclonable function (PUF) concept is based on the idea that, even if a set of several integrated circuits is created through an identical manufacturing process, each circuit is actually slightly different due to normal manufacturing variability. This variability can be used as a unique device fingerprint based on hardware. Examples of silicon PUF sources include logic race conditions [38, 71], Rowhammer behavior [17], and SRAM initialization data [39, 40]. Ruhrmair et al. [63] defined a fingerprint as "a small, fixed set of unique analog properties", and explain that the fingerprint should be measured quickly and preferably by an inexpensive device. In this work the GPU is used as a PUF, and our challenge is how to successfully capture the PUF behavior while using the limited APIs available to a web browser.

Fingerprint Spoofing. Fingerprint spoofing is a tactic used by adversaries when impersonating a device. Lin et al. [52] show that phishing adversaries can steal a user's browser fingerprint along with the user's credentials, and then log in with the victims's credentials while replaying the user's browser fingerprint to the website in order bypass 2FA. They show that this attack works on various websites, including banks and a credit card company. Campobasso et al. [24] analyze a real-world service, supported by a criminal infrastructure, that controls the supply chain of maliciously-acquired user profiles and offers these user profiles for sale. They show that this service stole more than 260 thousand user profiles. In addition to user credentials, these profiles were observed to contain browser fingerprints and other information, indicating that fingerprint spoofing may be used in the field by criminals. Sánchez-Rola et al. [65] found that one in four phishing pages adopt some form of fingerprinting. Liu et al. [53] design and implement the *Gummy Browsers* attack, which can successfully spoof a wide variety of fingerprinting features. They are able to fool FP-Stalker over 90% of the cases as adversaries, while still being able to track over 90% of benign users.

Microarchitectural Attacks on GPUs. Microarchitectural attacks are defined as a class of attacks that compromise the security guarantees of a system by exploiting the implementation characteristics of its underlying hardware [13]. Prior works have already shown how the GPU's hardware characteristics can be exploited by attackers. For example, Naghibijouybari et al., and later Ferguson et al., showed how OpenGL and WebGL code running on a GPU can fingerprint websites, track user activities, and infer keystroke timings through microarchitectural side-channel attacks [34, 58]. Frigo et al. showed how the Rowhammer vulnerability can be effectively exploited on GPUs, leading to an end-to-end mobile system compromise [36]. Finally, Wei et al. showed how the context-switching side channel can be used to extract fine-grained structural details of a deep-learning model running on a GPU shared between the attacker and the victim [77].

Common to all of these works is their focus on *security*. DrawnApart, in contrast, is the first work to use GPU microarchitectural variations in the *privacy* context, by showing how these variations can be exploited to perform browser fingerprinting and tell apart otherwise-identical devices.

Differences from the conference version. DrawnApart was first introduced in our NDSS paper [40], where we focused on its privacy implications by evaluating the technique in a controlled lab setting and by integrating a one-shot learning variant of DrawnApart into a state-of-the-art stateless tracking algorithm in the wild. In this journal

version, we complement these results with a third, security-motivated study that leverages the same non-deterministic GPU behavior as a defensive signal against fingerprint spoofing. Specifically, we consider a threat model in which attackers replay deterministic fingerprints to suppress risk-based 2FA challenges [52], and we show that inconsistencies between a device's DrawnApart trace and its claimed deterministic attributes can be detected by comparing the trace's embedding to reference embeddings associated with the claimed WebGL renderer string. This extension demonstrates that DrawnApart can be used to selectively waive 2FA for 19.4% of benign visits while still detecting 99% of spoofing attempts, improving security over approaches that waive 2FA based solely on deterministic fingerprint matches.

10 CONCLUSION

We introduced DrawnApart, an effective technique to create a browser fingerprint that relies on minor manufacturing variations in GPUs. Prior work on GPU microarchitectural side channels has already demonstrated that low-level GPU compute primitives can be used to extract fine-grained device properties and thus challenge the *security* of a system [36, 58]. Our work, however, is the first to show that GPU features can also be used to mount *privacy* attacks. We demonstrate this risk by presenting a concrete privacy-first instantiation of a PUF that can be used for browser fingerprinting. To the best of our knowledge, this is the first time GPU hardware features have been used to compromise privacy, presenting a significant expansion of the risk borne by this existing attack primitive. Our fingerprinting technique can tell apart devices that are completely indistinguishable by current state-of-the-art methods, while remaining robust to changing environmental conditions. Our technique works well both on PCs and mobile devices, has a practical offline and online runtime, and does not require access to any extra sensors such as the microphone, camera, or gyroscope. In a setting with diverse hardware and software configurations, DrawnApart is most effective when combined with traditional fingerprinting techniques such as FP-Stalker. Our technique can be also used to detect spoofing attempts aimed at bypassing two-factor authentication.

Processor designs are increasingly relying on massively parallel architectures to improve performance without breaking the physically-imposed constraints of power consumption and processor speed. As the capabilities of GPU hardware become increasingly exposed to untrusted web applications through APIs such as WebGPU, hardware and software designers must be aware of the risks to privacy raised by hardware fingerprinting, and take care to design software, drivers and hardware stacks in ways that protect user privacy.

Responsible Disclosure. We shared a preliminary draft of our paper with Intel, ARM, Google, Mozilla and Brave during June-July 2020 and continued sharing our progress with them throughout 2020 and 2021. In response to the disclosure, the Khronos group responsible for the WebGL specification has established a technical study group to discuss the disclosure with browser vendors and other stakeholders.

Artifact Availability. The JavaScript and GLSL collection code in the online, offline and GPU-based methods, the machine learning pipeline, as well as the Gen 3, Gen 4, Gen 8 and Gen 10 datasets, are all available in the following repository: https://github.com/drawnapart/drawnapart. The repository includes an interactive Python notebook, viewable over the web, that demonstrates classification over real data.

ACKNOWLEDGMENTS

This research has been supported by ANR-19-CE39-0007 MIAOUS; ANR-19-CE39-00201 FP-Locker; ANR-21-CE39-0019 FACADES; ANR-22-PECY-0002 IPoP; ANR-22-PECY-0009 REV projects; an ARC Discovery Early Career Researcher Award DE200101577; an ARC Discovery Project number DP210102670; the Blavatnik ICRC at Tel-Aviv University; the

Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through project number 560392681; Intel Corporation; and Israel Science Foundation.

We thank Gil Fidel, Anatoly Shusterman and Antoine Vastel for their advice and help. We are grateful to the BGU SISE technical support engineers Vitaly Shapira and Sergey Korotchenko for their help in setting up the evaluation test-beds. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Parts of this work were carried out while Yuval Yarom was affiliated with CSIRO's Data61.

REFERENCES

[1] [n.d.]. Adobe's end of life. https://www.adobe.com/products/flashplayer/end-of-life.html.

[2] [n.d.]. Brave's Shields. https://support.brave.com/hc/en-us/articles/360022973471-What-is-Shields-.

[3] [n.d.]. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa.

[4] [n.d.]. FingerprintJS. https://valve.github.io/fingerprintjs2/.

[5] [n.d.]. Ghostery. http:wwww.ghostery.com.

[6] [n.d.]. Privacy Badger. https://support.brave.com/hc/en-us/articles/360022973471-What-is-Shields-.

[7] [n.d.]. Samsung Remote Test Lab. https://developer.samsung.com/remotetestlab.

[8] [n.d.]. WebGL. https://www.khronos.org/webgl/.

[9] 2020. Changelog from v80.0.3987.163 to v.81.0.4044.92 – Chromium Git repository. https://chromium.googlesource.com/chromium/src/+log/80.0.3987.163..81.0.4044.92?pretty=fuller&n=10000.

[10] 2021. gpu_timing.cc – Chromium Code Search. https://source.chromium.org/chromium/chromium/src/+/master:ui/gl/gpu_timing.cc;l=309;drc=e5a38eddbdf45d7563a00d019debd11b803af1bb.

[11] 2021. Site Isolation – The Chromium Projects. https://www.chromium.org/Home/chromium-security/site-isolation.

[12] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juárez, Arvind Narayanan, and Claudia Díaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *CCS*. 674–689.

[13] Onur Aciiçmez. 2007. Yet another MicroArchitectural Attack: : exploiting I-Cache. In *CSAW*. ACM, 11–18.

[14] Ioannis Agadakos, Nikolaos Agadakos, Jason Polakis, and Mohamed R. Amer. 2020. Chameleons' Oblivion: Complex-Valued Deep Neural Networks for Protocol-Agnostic RF Device Fingerprinting. In *EuroS&P*. 322–338.

[15] Amani Al-Shawabka, Francesco Restuccia, Salvatore D'Oro, Tong Jian, Bruno Costa Rendon, Nasim Soltani, Jennifer G. Dy, Stratis Ioannidis, Kaushik R. Chowdhury, and Tommaso Melodia. 2020. Exposing the Fingerprint: Dissecting the Impact of the Wireless Channel on Radio Fingerprinting. In *INFOCOM*. 646–655.

[16] Furkan Alaca and Paul C. van Oorschot. 2016. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *ACSAC*. 289–301.

[17] Nikolaos Athanasios Anagnostopoulos, Tolga Arul, Yufan Fan, Christian Hatzfeld, André Schaller, Wenjie Xiong, Manishkumar Jain, Muhammad Umair Saleem, Jan Lotichius, Sebastian Gabmeyer, Jakub Szefer, and Stefan Katzenbeisser. 2018. Intrinsic Run-Time Row Hammer PUFs: Leveraging the Row Hammer Effect for Run-Time Cryptography and Improved Security. *Cryptography* 2, 3 (2018), 13.

[18] Zhongjie Ba, Sixu Piao, Xinwen Fu, Dimitrios Koutsonikolas, Aziz Mohaisen, and Kui Ren. 2018. ABC: Enabling Smartphone Authentication with Built-in Camera. In *NDSS*.

[19] Alex Berke, Badih Ghazi, Enrico Bacis, Pritish Kamath, Ravi Kumar, Robin Lassonde, Pasin Manurangsi, and Umar Syed. 2025. How Unique is Whose Web Browser? The role of demographics in browser fingerprinting among US users. *Proc. Priv. Enhancing Technol.* 2025, 1 (2025), 720–758.

[20] Károly Boda, Ádám Máté Földes, Gábor György Gulyás, and Sándor Imre. 2011. User Tracking on the Web via Cross-Browser Fingerprinting. In *NordSec*. 31–46.

[21] Hristo Bojinov, Yan Michalevsky, Gabi Nakibly, and Dan Boneh. 2014. Mobile Device Identification via Sensor Fingerprinting. *CoRR* abs/1408.1416 (2014). arXiv:1408.1416 http://arxiv.org/abs/1408.1416

[22] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32.

[23] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. 2016. Picasso: Lightweight Device Class Fingerprinting for Web Clients. In *SPSM@CCS*. 93–102. http://dl.acm.org/citation.cfm?id=2994467

[24] 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.

[25] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *NDSS*.

[26] W. B. Clarkson. 2012. *Breaking Assumptions: Distinguishing. Between Seemingly Identical Items Using Cheap Sensors.* Ph.D. Dissertation. Princeton.

[27] European Commission. [n.d.]. General Data Protection Regulation (GDPR). https://ec.europa.eu/info/law/law-topic/data-protection/eu-data-protection-rules_en.

[28] Sanorita Dey, Nirupam Roy, Wenyuan Xu, Romit Roy Choudhury, and Srihari Nelakuditi. 2014. AccelPrint: Imperfections of Accelerometers Make Smartphones Trackable. In *NDSS*.

[29] Matthijs Douze. 2022. Indexing 1M vectors. https://github.com/facebookresearch/faiss/wiki/Indexing-1M-vectors.

[30] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2025. The faiss library. *IEEE Transactions on Big Data* (2025).

[31] Antonin Durey, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2021. FP-Redemption: Studying Browser Fingerprinting Adoption for the Sake of Web Security. In *DIMVA*. 237–257. https://doi.org/10.1007/978-3-030-80825-9_12

[32] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *PETS*. 1–18.

[33] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *CCS*. 1388–1401. https://doi.org/10.1145/2976749.2978313

[34] Ethan Ferguson, Adam Wilson, and Hoda Naghibijouybari. 2024. WebGPU-SPY: Finding Fingerprints in the Sandbox through GPU Cache Attacks. In *AsiaCCS*. ACM.

[35] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. 2020. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. *PoPETs* 2020, 2 (2020), 499–518.

[36] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE SP*. 195–210.

[37] 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 *WWW*. 309–318. https://doi.org/10.1145/3178876.3186097

[38] Charles Herder, Meng-Day (Mandel) Yu, Farinaz Koushanfar, and Srinivas Devadas. 2014. Physical Unclonable Functions and Applications: A Tutorial. *Proc. IEEE* 102, 8 (2014), 1126–1141.

[39] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. 2007. Initial SRAM state as a fingerprint and source of true random numbers for RFID tags. In *Proceedings of the Conference on RFID Security*, Vol. 7. 01.

[40] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. 2009. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Trans. Computers* 58, 9 (2009), 1198–1210.

[41] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *IEEE SP*. 283–301. https://doi.org/10.1109/SP40001.2021.00017

[42] Hugo Jonker, Benjamin Krumnow, and Gabry Vlot. 2019. Fingerprint Surface-Based Detection of Web Bot Detectors. In *ESORICS*. 586–605. https://doi.org/10.1007/978-3-030-29962-0_28

[43] Google Kenneth Russell. [n.d.]. personal communication.

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE SP*. 1–19. https://doi.org/10.1109/SP.2019.00002

[45] D. Kristol and L. Montulli. 1997. *HTTP State Management Mechanism*. RFC 2109. RFC Editor.

[46] Tomer Laor, Naif Mehanna, Antonin Durey, Vitaly Dyadyuk, Pierre Laperdrix, Clémentine Maurice, Yossi Oren, Romain Rouvoy, Walter Rudametkin, and Yuval Yarom. 2022. DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting. In *Network and Distributed System Security Symposium*.

[47] Pierre Laperdrix, Gildas Avoine, Benoit Baudry, and Nick Nikiforakis. 2019. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *DIMVA*. 43–66. https://doi.org/10.1007/978-3-030-22038-9_3

[48] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser Fingerprinting: A Survey. *ACM Trans. Web* 14, 2 (2020), 8:1–8:33.

[49] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *IEEE SP*. 878–894.

[50] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srini Devadas. 2004. A technique to build a secret key in integrated circuits with identification and authentication applications. In *In Proceedings of the IEEE VLSI Circuits Symposium*. 176–179.

[51] Andy Liaw and Matthew Wiener. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[52] 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.

[53] 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.

[54] Ross McIlroy and Sami Kyostila. 2018. Chromium Commit: Clamp performance.now() to 100us. https://chromium.googlesource.com/chromium/src/+/a77687fd89adc1bc2ce91921456e0b9b59388120%5E%21.

[55] Markus Moenig. 2018. Issue 820891: WebGL2: EXT_disjoint_timer_query_webgl2 Failing in Beta of 65. https://bugs.chromium.org/p/chromium/issues/detail?id=820891.

[56] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. 2011. Fingerprinting information in JavaScript implementations. In *Proceedings of W2SP*, Vol. 2.

[57] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. *W2SP* (2012), 1–12.

[58] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael B. Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks are Practical. In *CCS*. 2139–2153.

[59] Gabi Nakibly, Gilad Shelef, and Shiran Yudilevich. 2015. Hardware Fingerprinting Using HTML5. *CoRR* abs/1503.01408 (2015). arXiv:1503.01408 http://arxiv.org/abs/1503.01408

[60] 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 *IEEE SP*. 541–555. https://doi.org/10.1109/SP.2013.43

[61] Tom Ritter. 2021. Comment 3 on Bug 1692609. https://bugzilla.mozilla.org/show_bug.cgi?id=1692609#c3.

[62] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. 2021. SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers. In *6th IEEE European Symposium on Security and Privacy (EuroS&P'21)*. Vienna, Austria. https://hal.inria.fr/hal-03215569

[63] Ulrich Rührmair, Srinivas Devadas, and Farinaz Koushanfar. 2012. Security based on physical unclonability and disorder. In *Introduction to Hardware Security and Trust*. Springer, 65–102.

[64] Kenneth Russell. 2020. Issue 859249: Extend WebGL 2.0 Compute flag expiry to M85. https://chromium.googlesource.com/chromium/src.git/+/96186af9c385db253bf85f06f1324a729684cb2f.

[65] Iskander Sánchez-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.

[66] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. 2018. Clock Around the Clock: Time-Based Device Fingerprinting. In *CCS*. 1502–1514.

[67] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *CVPR*. 815–823. https://doi.org/10.1109/CVPR.2015.7298682

[68] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security*. 247–267. https://doi.org/10.1007/978-3-319-70972-7_13

[69] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. 2021. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security*.

[70] Peter Snyder, Cynthia Bagier Taylor, and Chris Kanich. 2017. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *CCS*.

[71] G. Edward Suh and Srinivas Devadas. 2007. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *DAC*. 9–14.

[72] Ben L. Titzer and Jaroslav Sevcik. 2019. A year with Spectre: a V8 perspective. https://v8.dev/blog/spectre.

[73] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *USENIX Security*. 135–150. https://www.usenix.org/conference/usenixsecurity18/presentation/vastel

[74] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. FP-Stalker: Tracking Browser Fingerprint Evolutions. In *IEEE SP*. 728–741.

[75] Antoine Vastel, Walter Rudametkin, Romain Rouvoy, and Xavier Blanc. 2020. FP-Crawlers: Studying the Resilience of Browser Fingerprinting to Block Crawlers. In *MADWeb*.

[76] Jóakim von Kistowski, Hansfried Block, John Beckett, Cloyce Spradling, Klaus-Dieter Lange, and Samuel Kounev. 2016. Variations in CPU Power Consumption. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/2851553.2851567

[77] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In *DSN*. IEEE, 125–137.

[78] Stephan Wiefling, Markus Dürmuth, and Luigi Lo Iacono. 2020. More Than Just Good Passwords? A Study on Usability and Security Perceptions of Risk-based Authentication. In *ACSAC*. ACM, 203–218.

[79] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. 2019. Rendered Private: Making GLSL Execution Uniform to Prevent WebGL-based Browser Fingerprinting. In *USENIX Security*. 1645–1660.

[80] Jiexin Zhang, Alastair R. Beresford, and Ian Sheret. 2019. SensorID: Sensor Calibration Fingerprinting for Smartphones. In *IEEE SP*. 638–655. https://doi.org/10.1109/SP.2019.00072

[81] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. 2014. Acoustic Fingerprinting Revisited: Generate Stable Device ID Stealthily with Inaudible Sound. In *CCS*. 429–440.