DRAWNAPART: A Device Identification Technique based on Remote GPU Fingerprinting

Tomer Laor*
Ben-Gurion Univ. of the Negev
tomerlao@post.bgu.ac.il

Naif Mehanna*
Univ. Lille, CNRS, Inria
naif.mehanna@univ-lille.fr

Antonin Durey
Univ. Lille, CNRS, Inria
antonin.durey@univ-lille.fr

Vitaly Dyadyuk
Ben-Gurion Univ. of the Negev
vitalyd@post.bgu.ac.il

Pierre Laperdrix
Univ. Lille, CNRS, Inria
pierre.laperdrix@univ-lille.fr

Clémentine Maurice
Univ. Lille, CNRS, Inria
clementine.maurice@inria.fr

Yossi Oren
Ben-Gurion Univ. of the Negev
yos@bgu.ac.il

Romain Rouvoy
Univ. Lille, CNRS, Inria / IUF
romain.rouvoy@univ-lille.fr

Walter Rudametkin
Univ. Lille, CNRS, Inria
walter.rudametkin@univ-lille.fr

Yuval Yarom
Univ. of Adelaide
yval@cs.adelaide.edu.au

Abstract—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.

In this paper, we report on a new technique that can significantly extend the tracking time of fingerprint-based tracking methods. Our technique, which we call DRAWNAPART, is a new *GPU fingerprinting* technique that identifies a device from the unique properties of its GPU stack. Specifically, we show that variations in speed among the multiple execution units that comprise a GPU can serve as a reliable and robust device signature, which can be collected using unprivileged JavaScript. We investigate the accuracy of DRAWNAPART under two scenarios. In the first scenario, our controlled experiments confirm that the technique is effective in distinguishing devices with similar hardware and software configurations, even when they are considered identical by current state-of-the-art fingerprinting algorithms. In the second scenario, we integrate a *one-shot learning* version of our technique into a state-of-the-art browser fingerprint tracking algorithm. We verify our technique through a large-scale experiment involving data collected from over 2,500 crowd-sourced devices over a period of several months and show it provides a boost of up to 67% to the median tracking duration, compared to the state-of-the-art method.

DRAWNAPART makes two contributions to the state of the art in browser fingerprinting. On the conceptual front, it is the first work that explores the manufacturing differences between identical GPUs and the first to exploit these differences in a privacy context. On the practical front, it demonstrates a robust technique for distinguishing between machines with identical hardware and software configurations, a technique that delivers practical accuracy gains in a realistic setting.

I. 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 [50]. Recent regulations restrict and supervise the acquisition of private data by websites [4, 31], 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 [52], even in a collection spanning millions of fingerprints [43]. Surveying the Internet

---

*Both authors are considered co-first authors.

demonstrates that browser fingerprinting techniques are prevalent and used by many websites, no matter their category or ranking [38, 40, 59].

A significant difficulty of fingerprint-based tracking is that browser fingerprints evolve. As shown by Vastel et al. [73], 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.

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 [44, 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. Extracting a hardware fingerprint from a browser, however, is far from trivial—since the attacker has little control. In particular, the attacker can only interact with the system through unprivileged JavaScript code and WebGL graphics primitives—the attacker has no control over the runtime environment of the system, including background processes and simultaneous user activity—and the attacker has very limited exposure to the system, making classical machine learning pipelines that rely on long training phases all but useless. 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 evaluate DRAWNAPART in two 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 integrate the technique into Vastel et al.'s state-of-the-art fingerprinting algorithm from IEEE S&P 2018 [73], which uses machine learning to link browser fingerprint evolutions. We show that the median tracking duration is improved by up to 66.66% once we add the DRAWNAPART fingerprint.

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 III).
- 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 V).
- We integrate DRAWNAPART into Vastel et al.'s fingerprinting algorithm and show, through a large-scale crowdsourced 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 VI).
- We suggest possible countermeasures against our fingerprinting technique, and discuss their advantages and drawbacks (Section VII-B).

II. BACKGROUND

A. 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 [27, 48, 74], or support authentication, where the fingerprint is used in addition to a traditional authentication mechanism [20, 51]. A destructive use might involve tracking users without consent [17, 40]. 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 [24, 39, 57, 58]. 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.

B. 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*

---

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

(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, while the Mali-G72 GPU—integrated into the Samsung Exynos 9810 chipset used in Galaxy S9, S9+, Note9, and Note10 Lite devices—includes 18 EUs.

Web Graphics Library (WebGL) is a cross-platform API for rendering 3D graphics in the browser [12]. 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 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).

III. GPU FINGERPRINTING

A. Motivation

Similar to past work [39, 52], 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. As we show experimentally, this additional distinguishing power can considerably enhance the tracking capabilities of existing fingerprinting methods. To do so, we incorporate techniques similar to the arbiter-based *Physically Unclonable Function* (PUF) concept of Lee et al. [53]. 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.

B. Design

With unfettered access to the GPU, an adversary could measure the speed of each EU and use those measurements as a fingerprint. However, websites only have limited access to the GPU through the JavaScript and WebGL APIs. WebGL provides a high-level abstraction that makes it a challenge to target specific EUs and to time computations accurately.

We overcome this challenge by using short GLSL programs executed by the GPU as part of the vertex shader (cf. Section II-B). We rely on the mostly predictable job allocation in the WebGL software stack to target specific EUs. We observe that, when allocating a parallel set of vertex shader tasks, the WebGL stack tends to assign the tasks to different EUs in a non-randomized fashion. This allows us to issue multiple commands that target the same EUs. Finally, instead of measuring specific tasks, we ensure that the execution time
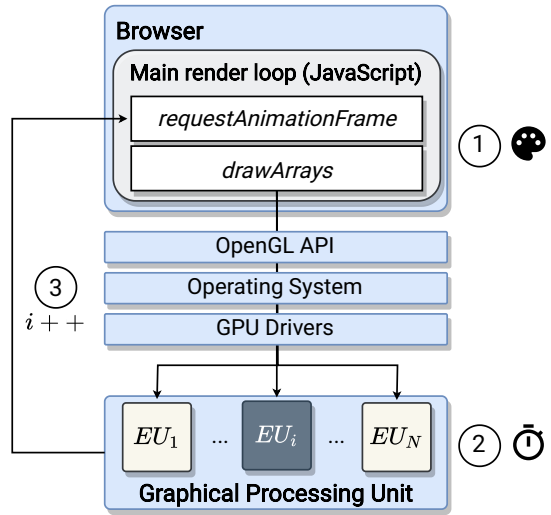

Fig. 1. Overview of our GPU fingerprinting technique: (1) points are rendered in parallel using several EUs; (2) the EU drawing point i executes a stall function (dark), while other EUs return a hard-coded value (light); (3) the execution time of each iteration is bounded by the slowest EU.

of the targeted EU dominates the execution time of the whole pipeline. We do so by assigning the non-targeted EUs a vertex shading program that is quick to complete, while assigning the targeted EUs tasks whose execution time is highly sensitive to the differences among individual EUs. As shown in Figure 1, our fingerprint is created by executing a sequence of drawing operations. We measure the time to draw a sequence of points with carefully chosen shader programs. The technique consists of three main steps:

Render. We instruct the WebGL API to draw a number of points in parallel. Points are the simplest object that WebGL can draw, and each consists of only a single vertex. Using points minimizes the noise from the pipeline and its interference with our technique. The position of each point is determined by an attacker-controlled vertex shader.

Stall. For most points, the attacker-controlled vertex shader returns a hard-coded value. For a specific subset of the points the shader applies a function, which we call a *stall function*, to compute the point's position. The manner in which the entire graphics stack distributes the points to be drawn to the EUs allows us to influence which EU is chosen to run the stall function. It takes much longer to compute the position with the stall function than the hard-coded value. As a result, the time needed to render the entire set of points corresponds to the time taken by the EUs running the stall function.

Trace Generation. We execute the drawing command several times, each time selecting a different vertex to stall. For each execution, we store the time taken. The fingerprint output by our technique is therefore a vector, named a *trace*, which contains the sequence of timing measurements.

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.

C. Implementation

We now describe the implementation of each design step.

Render. The WebGL API exposes the `drawArrays()` function, which allows dispatching multiple drawing operations in parallel to the GPU. We invoke `drawArrays()` several times, each time rendering multiple points in parallel. Listing 1 describes our main render loop. We execute the rendering process by calling `drawArrays` (line 5). For each iteration, we save the time to execute `drawArrays` into the `trace` array. We evaluated several ways of measuring the rendering time, as explained further in Section V-A. Briefly put, the **onscreen** measurement method executes a relatively small number of computationally intensive operations, while the **offscreen** and **GPU** measurement methods execute a larger number of less computationally intensive operations. The full source code for these settings can be found in our artifact repository, as listed in Section IX. After `point_count` iterations, the code sends the `trace` array to our back-end server (line 15), and terminates the loop.

```
1   function render_loop() {
2     if (point_index < point_count) {
3         // Stall the current point
4         gl.uniform1i(shader_stalled_point_id,
              point_index);
5         gl.drawArrays(gl.POINTS,0,point_count);
6         // Save the rendering time
7         var dt = performance.now() - prev_time;
8         prev_time = performance.now();
9         trace.push(dt);
10        // Prepare to stall the next point
11        point_index++;
12        requestAnimationFrame(render_loop);
13    } else {
14      // Finish and send the trace to the server
15      send_trace();
16    }
17  }
```

Listing 1.   Main Render loop, onscreen setting (JavaScript).

Stall. In the current implementation of WebGL, a single call to `drawArrays()` generates multiple drawing operations in the underlying graphics API, which appear to assign vertices to EUs in a deterministic order during vertex processing. The operations are differentiated by a global variable, named `gl_VertexID`. This special variable is an integer index for the current vertex, intrinsically generated by the hardware in all of the graphics APIs used to implement WebGL as it executes `gl.drawArrays`. We created a vertex shader in GLSL that examines the `gl_VertexID` identifier, and executes a computationally intensive *stall function* only if it matches an input variable named `shader_stalled_point_id` provided by the JavaScript code running on the CPU. Listing 2 describes the vertex shader code.

In the onscreen setting, the vertex shader checks if `shader_stalled_point_id` equals `gl_VertexID`. In the offscreen and GPU settings, the vertex shader treats `shader_stalled_point_id` as a bit mask and checks if

```
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 2.   Vertex shader with stall function, onscreen setting (GLSL).
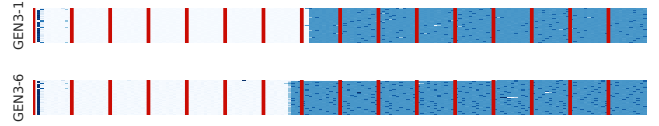

Fig. 2.   Raw traces from two different GEN 3 devices.

bit `1 << gl_VertexID` is set. In both cases, if the point is selected the vertex shader program executes the stall function (line 5). Otherwise, the shader exits quickly.

Trace Generation. By executing this parallel drawing operation multiple times, each with a different value for `shader_stalled_point_id`, we iterate over the different EUs and measure the relative performance of each. The output is a trace of multiple timing measurements, corresponding to the time taken by the targeted EU to draw the scene.

D. Raw Traces

Before evaluating DRAWNAPART, we tested whether we can visually distinguish devices. Figure 2 shows traces collected from two GEN 3 devices. We collect 50 traces from each device, each trace consisting of 176 measurements of 16 points. The measurements are divided into 16 groups of 11, where in each group we stall a different point. The color of a point indicates the rendering time, ranging from virtually 0 (white) to 90 ms (blue). Red vertical bars indicate group boundaries. As we can see, the rendering time in the first half of the traces is significantly faster than in the second half. Moreover, while there are some timing variations in the traces of the same device, the traces display patterns that are distinct between devices, allowing us to distinguish them.

IV.   EVALUATION OVERVIEW

A. 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. Second, the attacker can tailor the attack to the particular class of devices to be discriminated, and thus choose optimal parameters for the target hardware. Third, the workload on the target machines is controlled, minimizing measurement noise. Finally, the attacker is not concerned with detectability or compatibility, and can run an experiment that takes a long time, that uses partially supported hardware features, or that is noticeable to the user.

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, including the browser version, screen dimensions, HTTP headers, and other similar attributes. 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 [73]. 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.

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 VI, 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.

B. 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 evaluated several classical machine learning models suitable for this task, including tree-based classifiers, k-Nearest Neighbors classifiers, Linear Discriminant Analysis, and Support Vector Machines. We ultimately chose to use the Random Forest ensemble classification algorithm [26, 54], 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 IV-A. 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 VI.

V. LAB SETTING

The objective of the lab setting is to discover DRAWN-APART'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 I 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 an Apple M1 chip, running MacOS X Version 11.1. Other than the GEN 10 devices, which had discrete Nvidia GTX1650 GPUs, all desktops used integrated graphics. For

TABLE I. 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

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 [10]. All the mobile devices were Android-based and featured Samsung Exynos CPUs and Mali GPUs.

Comparison With Prior Fingerprinting Techniques. Before evaluating our technique, we reproduced and tested several state-of-the-art web-based fingerprinting techniques.

UniqueMachine, presented by Cao et al. at NDSS 2017 [28], collects a "browser fingerprint", with mutable properties such as window size and IP address, and a more permanent "computer fingerprint". The UniqueMachine website offers a demo that outputs both fingerprints as 32-character hashes. We collected the fingerprints of all of the computers in our GEN 3, GEN 4, GEN 8, and GEN 10 corpora using UniqueMachine, and confirmed that all computers in the same corpus were assigned the same computer fingerprint. Interestingly, the GEN 4 and GEN 10 PCs shared the same computer fingerprint despite having different hardware configurations.

Fingerprint JS (FPJS) is a commercial API offering "browser fingerprinting as a service". The paid-for version, called FPJS Pro, claims to provide "unparalleled accuracy, ease of use, and security" [6]. FPJS Pro outputs a 20-character hash. The website provides a demo of FPJS Pro. We collected the fingerprints of all computers in our GEN 3, GEN 4, GEN 8, and GEN 10 corpora using the demo website. In the GEN 3 dataset, all but one computer had the same fingerprint. Similarly to UniqueMachine, all of the computers in the GEN 4 and GEN 10 corpora had identical FPJS fingerprints. Finally, FPJS divided the GEN 8 corpus into three clusters: two clusters with seven computers each, and the final cluster with one computer.

Clock around the Clock, proposed by Sánchez-Rola et al. at CCS 2018 [66], is an alternative to GPU-based fingerprinting. This method is designed to exploit "small, but measurable, differences in the clock frequency" by measuring the precise execution times of a series of CPU-intensive operations. To calculate the fingerprint, the computer invokes the cryptographic random number generator `crypto.getRandomValues` 1,000 times for 50 different input sizes, then generates a vector of the most common timing

value, or mode, for each of the input sizes. We reproduced the web-based variant of the method, and tested it on our GEN 4 corpus. We found that the modes did not contain any data useful for fingerprinting. This is likely because since July 2018 Chrome contains countermeasures designed to prevent fine-grain timing measurements, as part of the wider fallout of the Spectre attacks [29, 60, 62, 77]. All our measurements returned either zero or five microseconds (with some added randomness). We conclude that, currently, the method presented by Sánchez-Rola et al. is not practical.

A. Tuning the Trace Parameters

We search for the parameter settings that provide the optimal accuracy gain for the different hardware configurations.

Stall Function Operator Selection. Each model and generation of GPU has a different micro architecture. For example, the third-generation Intel integrated GPU has a single arbiter, which dispatches tasks to all EUs, while fourth-generation GPUs adopt a hierarchical micro-architecture with multiple arbiters. Intel GPUs also have *Advanced Math (AM) Units*, which are tasked with executing less common operations such as trigonometric operations. The amount and location of these AM units differs among GPU generations, and even within different GPU types from the same generation. The design of GPUs by Nvidia, ARM and Apple is obviously different as well. We hypothesize that, due to these differences, the accuracy gain provided by our method will vary, depending both on the choice of stall functions and target hardware. To test this, we evaluated a representative set of operators, including trigonometric operations, logical bit-wise operations, and general floating-point operations. The set of operators selected can be found in Appendix A.

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, 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. Even though the canvas element is on screen, it can be made zero-size or invisible via styling, making the fingerprinting operation invisible to the user. Collecting the fingerprint does cause a noticeable slowdown for the user since it runs in the browser's main context.

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. At the time of writing, `OffscreenCanvas` is supported on Chrome browsers, hidden behind a flag on Firefox, and partially supported in the Technical Preview build of Safari.

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 [42]. As a result, support for this timer was disabled in Chrome version 65. However, with the introduction of Site Isolation [16], 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 [63], the GPU-side timer offers microsecond resolution with no jitter even on the most modern versions of Chrome [15]. 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.

Number Of Points To Render. Our fingerprinting scheme relies on multiple iterations of a drawing command, where each iteration exercises a certain subset of the EUs while leaving the other EUs idle. The number of iterations and the time each iteration takes to run will determine the total execution time. However, it is reasonable to assume that capturing more data will provide better accuracy, and that relatively long workloads will mitigate the impact of the low-resolution timers available through JavaScript. We ran two experiments to capture this

trade-off. The first was run in the onscreen setting, using the GEN 3 corpus. The frame rate requirement of the onscreen setting limits each iteration to at least 16 ms, as explained above. The second experiment was run in the offscreen setting using the GEN 4 corpus. This setting allowed us to use much shorter workloads and to increase the number of iterations that can be run in a reasonable time period. Thus, instead of assigning the stall function for each point only once per iteration, we tried all 2^n possible subsets of the set of points, allowing us to measure the *contention* between EUs, as well as their individual speeds.

B. Results

Table I 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. Specifically, in the onscreen setting, the `mul` operator delivered the best accuracy gains for the GEN 3 and GEN 4 corpora, while `exp2` was the best performer for the GEN 8 corpora, as described in more detail in Appendix A. The different mobile device corpora, which were also evaluated in the onscreen setting, also had different optimal operators: `pow` for Galaxy S8/S8+ and Galaxy S9/S9+, `atanh` for Galaxy S10e/S10/S10+ and `mul` for Galaxy S20/S20+/S20 Ultra.

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 I, 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.

Accuracy vs. Capture Time.   Figure 3 shows the accuracy gain as a function of trace capture time, both for the GEN 3 corpus using the onscreen collection method, and for the GEN 4 using the offscreen collection method. As the Figure shows, the accuracy gain of both methods approaches its optimal point when samples are collected for around 2 seconds. This is reached after about 80 iterations in the onscreen method and 1024 iterations in the offscreen method.
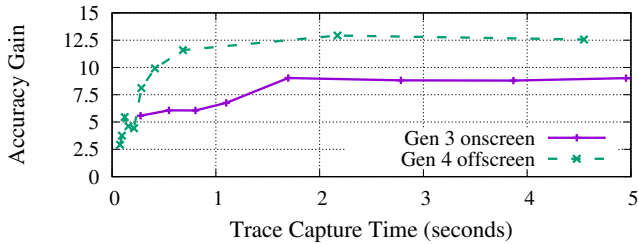

Fig. 3.   Accuracy gain as a function of trace capture time

Swapping Hardware.   To reinforce our claim that the classification results are due to differences in the behavior of the GPUs, and not due to some residual differences among the computers, we selected two GEN 3 computers, physically swapped their hard drives, and re-ran the fingerprinting classifier. As expected, the fingerprinting classifier was not misled by the hard disk transplant, and was still able to label each of the two computers according to their CPU. Next, we returned the hard drives to their original locations, and physically swapped the CPUs with integrated graphics of the two systems. As expected, the classifier followed the transplanted CPU, even though all other hardware was unmodified.

C. Evaluation on Additional Browsers.

We collected and evaluated traces from 16 devices from the GEN 4 corpora using multiple additional browsers: Brave browser [2] (version 81.0.4044.113), Edge [5] (version 96.0.1054.43), Opera [8] (version 82.0.4227.23) and Yandex browser [13] (version 21.11.3.927), all using the *offscreen* method. The accuracy showed a significant improvement over the base rate, which lies at 6.25%, with Edge, Brave, Opera and Yandex, delivering accuracies of 34.6±0.6%, 31.0±0.3%, 31.6±0.7%, and 31.1±0.3%, respectively.

We evaluated the stability of DRAWNAPART over 21 devices of the GEN 4 corpora for an extended period of time. We collect data for both Chrome and Firefox. For Chrome, we use the *onscreen* and *offscreen* methods. For Firefox, which does not currently support the *offscreen* method, we are limited to the *onscreen* method. We also chose to stall the EU for twice

as many operations under Firefox, compared to Chrome, to account for the lower timer resolution found in Firefox.

For 24 days, we repeatedly launched the browser, collected traces for 20 minutes using the *offscreen* method and for 40 minutes using the *onscreen* method, then quit the browser and idled for 4 hours. The first 4 cycles were used to train the Random Forest classifier, while the remaining cycles over the experiment's 24 days were used to evaluate its performance. The results are summarized in Figure 4 and show the accuracy to be above the base rate for each point in time. We observed that the *offscreen* method yields slightly higher accuracy than *onscreen*, and that the accuracy of both methods on Chrome slightly decay over time, while the accuracy of the *onscreen* method on Firefox remains stable. Finally, the accuracy in this experiment is lower compared to the results reported in Table I. It is possibly due to repeatedly restarting the browser over the course of the experiment, as we discuss in Section VII-C.
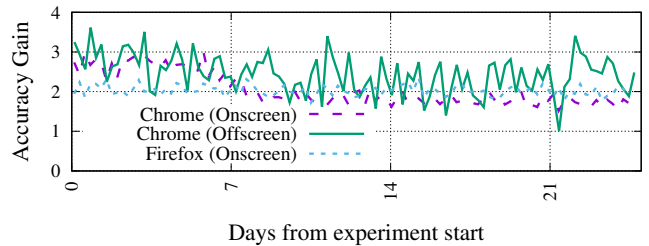

Fig. 4.   Additional Browsers – Lab Evaluation

D. Summary

Our results show that DRAWNAPART can tell apart identical computers in a controlled lab setting. Our next objective was to 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.

VI.   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 reidentifying a device among

others. In Section VI-B, 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 [39]. Vastel et al. developed two algorithms to track evolutions and link fingerprints that belong to the same device [72]. In Section VI-D, 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 Appendix B as well as the DRAWNAPART traces.

A. 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 [52]. 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 is supported by all desktop versions of Chrome. The onscreen method was not selected as it causes slowdowns, and the GPU method was not selected since it is not supported by the Google SwiftShader renderer. 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. This fingerprint takes a median time of 1.6 seconds to run. It is collected by the extension using a worker thread, without affecting the user's interactions with the browser. To increase our trace count, we repeated each collection seven times, for a median total run time of approximately 12 seconds. 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 Appendix B, 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 $1\mathsf{MP}_{65}$. The rest of 1MP will be referred to as $1\mathsf{MP}_{rest}$ 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.

B. 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 space, which express the distance between traces. We begin the process by creating a *Convolutional Neural Network* (CNN)-based multinomial classifier. The structure we selected for the classifier is inspired by Picek [61], and includes N convolution blocks followed by a flatten layer, a dense layer, another L2-normalized dense layer without activation, and concluding with a fully connected layer with softmax activation. Each convolution block contains a convolution layer, a dropout layer, and an average pooling layer. We used scikit-optimize's Bayesian optimization [11] to search for the best parameters, as described in Appendix C, using 80% of the traces in $1\mathsf{MP}_{65}$ for training, and the remainder of $1\mathsf{MP}_{65}$ for validation. The parameter search took 48 hours on a server with four NVIDIA GEFORCE RTX 2080 Ti GPUs, two Intel Xeon Silver 4110 CPUs, and 128 GiB of RAM. The run yielded 79 valid neural networks. The best network 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. Triplet loss minimizes the distance between an anchor and a positive, both of which have the same label, and maximizes the distance between the anchor and a negative of a different label. Semi-hard triplet loss means that we only use triplets that have a negative that is farther from the anchor than the positive, but still produces a positive loss [67]. We took our trained classification model, removed its last layer, and trained it again for 30 epochs on the same dataset as before, this time with a bigger batch size of 1024 preprocessed traces and with semi-hard triplet loss. Batch size is important to the triplet mining process since we need sufficient examples in the batch to find enough semi-hard triplets. 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. We note that we optimized for the accuracy of the classification model, instead of the 1-Nearest Neighbor, to reduce the running time of our parameter search.

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. On this subset, the base rate is 1.00% for top-1 accuracy, 3.51% for top-5 accuracy and 6.15% for top-10 accuracy. 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 $1MP_{rest}$. On this subset, the base rate is 1.22% for top-1 accuracy, 4.42% for top-5 accuracy and 7.2% for top-10 accuracy. 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 $2MP$. On this subset, the base rate is 0.64% for top-1 accuracy, 2.78% for top-5 accuracy and 4.38% for top-10 accuracy. The results in Table II 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 of the 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 collections as possible. This evaluation is harder than the previous one

TABLE II.   STANDALONE PERFORMANCE OF DRAWNAPART IN THE WILD USING THE RANDOM SPLIT (RS) AND k-SHOT METHODS

Evaluation Method	Accuracy (*Base rate*)		
(Dataset)	**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%*)

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 II. 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 correlations between traces in the same collection, we used only the first trace in the collections that we sampled from. It means that we measured the distance between traces from different collections only. Figure 5 presents the probability density of the different distributions. As the figure shows, embeddings from the same device get a lower Euclidean distance compared to embeddings from different devices, even if the device has the same GPU. 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 or an element correlated with 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. This is a strong property, we use it in the next section to improve FP-STALKER.

C. Evaluation on additional browsers in the wild.

While approximately 93.8% of the traces found in our in-the-wild dataset $2MP$ come from users running the Google Chrome browser, some users submitted traces using other Chromium-based browsers. We isolated non-Chrome users by filtering the traces according to their *user-agent*, and analyzed the effectiveness of our standalone machine learning pipeline on these browsers as well. The non-Chrome traces came from users running Edge, Opera and Yandex, which represented 5%, 0.7% and 0.5% of the traces respectively. We run the evaluation pipeline described in Section VI-B for each browser,
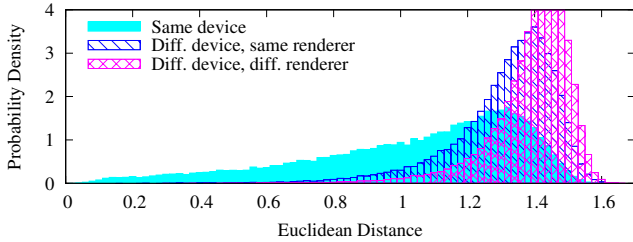
Fig. 5. Performance of the DRAWNAPART embedding function. A Euclidean distance below 0.65 indicates that the traces are likely to be from the same device.

independently. Our results show that the standalone pipeline's accuracy for Edge, Opera and Yandex is 52.6%, 79.3%, and 89.7%, respectively. The smaller amount of traces in this subset of the data results in a higher base rate when compared to the entire 2MP dataset—3% for Edge, 17.9% for Opera, and 27.6% for Yandex. These results, with the lab setting results, indicate that our fingerprinting technique identifies browsers from multiple vendors. More details can be found in Appendix D.

Summary. The results of the standalone evaluation, as summarized in Table II, show a significant improvement over the base rate, demonstrating that DRAWNAPART is effective on its own. 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.

D. Integrating DRAWNAPART with FP-STALKER

FP-STALKER is the state-of-the-art fingerprint linking algorithm [73]. 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 or have been spoofed and cannot be normally found in the wild, 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. Appendix E lists the linking algorithm.

Improving The Algorithm. As mentioned in Section VI-B, the output of the embedding network consists of 256 L2-normalized points that allow us to use a Euclidean distance to compute the similarity between embeddings. Figure 5 shows that the Euclidean distance is efficient, to an extent, in differentiating devices. Based on the results obtained in Section VI-B, 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 5 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 used the previously generated averaged embeddings to compute the cosine similarity of the two compared fingerprints. 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 fingerprint as similar to the one being compared without further steps. The algorithm with the DRAWNAPART additions, is available in Appendix B.

Choosing The Epsilon Threshold. We chose the threshold by performing an analysis over similar and different devices on the train dataset. 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 5, was chose the cosine similarity for FP-STALKER because it is bounded by a more natural interval of $[-1; 1]$. Our experiments showed that our threshold on the cosine similarity yielded better results than our Euclidean distance threshold. 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 VI-A. We first trained the Random Forest model on fingerprints in the 1MP subset. We then executed the lambda

TABLE III.   AVERAGE TRACKING TIME BY COLLECTION PERIOD

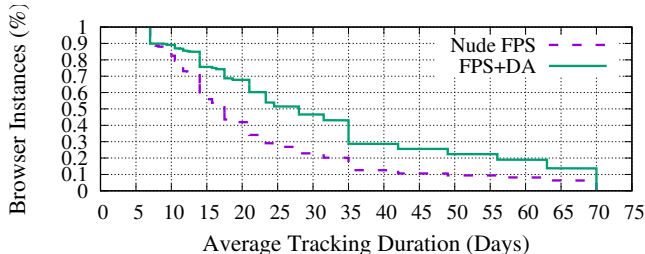| Collection Period | Tracking duration in days | | Improvement |
	Nude FPS	FPS+DA	
2 days	17	26	+52.94%
3 days	17.25	25.5	+47.82%
4 days	17	28	+64.70%
5 days	17.5	27.5	+57.14%
6 days	18	30	+66.66%
7 days	17.5	28	+60.00%

Fig. 6.   Differences in Average Tracking Time between FP-STALKER (Nude FPS) and FP-STALKER with DRAWNAPART (FPS+DA)

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 III 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 days. Figure 6 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.

VII. DISCUSSION

A. Ethical Concerns

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.

B. 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 [3] and extensions, such as Ghostery [7] or Privacy Badger [9]. However, filter lists against trackers and fingerprinting have been shown to lack exhaustiveness [41, 47].

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 [72]. Wu et al. [76] 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. [76].

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 [76]. 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.

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].

C. Limitations and Insights

Experimental Limitations. The in the wild, crowd-sourced experiments demonstrate that DRAWNAPART can work successfully in a variety of conditions that are not under the attacker's control. However, our lab experiments only cover a limited set of conditions. Specifically, we only evaluated the impact of temperatures between 26.4 °C and 37 °C, demonstrating no impact on the results. Hence, we cannot preclude the possibility that temperatures outside this range do not affect the results. Similarly, our lab experiments do 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 in normal operating conditions.

Approach Limitations. We evaluate the effect of device restarts on fingerprinting accuracy by training a model on the GEN 3 devices, and testing the model against traces collected after rebooting the devices. We obtain an overall accuracy of 50.3%. We observe that the accuracy drop is not uniform. That is, some devices maintain stable fingerprints across restarts, whereas the fingerprints of others change significantly each restart. We note that we do not track reboots in our in-the-wild experiments. Hence, these already account for the potential accuracy drop associated with restarts.

We evaluate our technique across ten Chrome versions, from 80.0.3987.116 to 81.0.4044.138. These ten versions consist of two groups: the v80 group which includes six minor versions, and the v81 which includes four minor versions. We train our classifier on the latest v80 version (80.0.3987.163) and test all ten versions. We obtain an accuracy of around 90% on all v80 versions, but significantly lower accuracy, of around 60%, when we test the trained model on v81. We hypothesize some changes in Chrome between v80 and v81 affected the entire WebGL stack. Observing the changelog for the Chromium code repository reveals more than 10,000 commits between the two versions with several hundreds affecting the GPU and the WebGL API [14]. An additional experiment we conducted show that an attacker with a limited trace capture budget can maintain an up-to-date classification model by training a combined model with traces from multiple versions and obtaining a consistently high accuracy of 90±% across all ten versions.

D. 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 propsals are that: 1) DrawnApart might be uncovering differences in power consumption. A study by von Kistowski et al. [75] 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 Compute was integrated into Chrome but disabled in 2020 [65], and its development has been subsumed by WebGPU [49]. WebGPU is currently under active development, and is not supported in the stable edition of any browser, but preliminary implementations can be found in the canary versions of Firefox, Chrome, and Edge.

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.

VIII. RELATED WORK

Web-based Fingerprinting. Eckersley [39] 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 [27, 48, 74], or help to protect sign-in processes [20, 51]. 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 [39, 52], font enumeration [59], audio rendering [40], 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 [43]. However, they possess additional fingerprinting sources such as sensors [25, 33, 36, 37, 78], microphones [23, 30, 34, 35, 79]

and cameras [22, 32]. Manufacturing variations can also manifest as differences in the radio frequency (RF) behavior of networked devices [18, 19]. 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 [44, 71], Rowhammer behavior [21], and SRAM initialization data [45, 46]. Ruhrmair et al. [64] 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.

IX. CONCLUSION

We introduced an effective technique to create a browser fingerprint that relies on minor manufacturing variations in GPUs. To the best of our knowledge, this is the first time hardware features have been used to challenge privacy in this context. 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.

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.

BIBLIOGRAPHY

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

[2] "Brave," https://brave.com/.

[3] "Brave's shields," https://support.brave.com/hc/en-us/articles/360022973471-What-is-Shields-.

[4] "California Consumer Privacy Act (CCPA)," https://oag.ca.gov/privacy/ccpa.

[5] "Edge," https://www.microsoft.com/en-us/edge.

[6] "FingerprintJS," https://valve.github.io/fingerprintjs2/.

[7] "Ghostery," http:www.ghostery.com.

[8] "Opera," https://www.opera.com/.

[9] "Privacy badger," https://support.brave.com/hc/en-us/articles/360022973471-What-is-Shields-.

[10] "Samsung Remote Test Lab," https://developer.samsung.com/remotetestlab.

[11] "scikit-optimize: Sequential model-based optimization in python," https://scikit-optimize.github.io/stable/.

[12] "WebGL," https://www.khronos.org/webgl/.

[13] "Yandex browser," https://browser.yandex.ru/beta/.

[14] "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, 2020.

[15] "gpu_timing.cc – chromium code search," https://source.chromium.org/chromium/chromium/src/+/master:ui/gl/gpu_timing.cc;l=309;drc=e5a38eddbdf45d7563a00d019debd11b803af1bb, 2021.

[16] "Site isolation – the chromium projects," https://www.chromium.org/Home/chromium-security/site-isolation, 2021.

[17] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *CCS*, 2014, pp. 674–689.

[18] I. Agadakos, N. Agadakos, J. Polakis, and M. R. Amer, "Chameleons' oblivion: Complex-valued deep neural networks for protocol-agnostic RF device fingerprinting," in *EuroS&P*, 2020, pp. 322–338.

[19] A. Al-Shawabka, F. Restuccia, S. D'Oro, T. Jian, B. C. Rendon, N. Soltani, J. G. Dy, S. Ioannidis, K. R. Chowdhury, and T. Melodia, "Exposing the fingerprint:

Dissecting the impact of the wireless channel on radio fingerprinting," in *INFOCOM*, 2020, pp. 646–655.

[20] F. Alaca and P. C. van Oorschot, "Device fingerprinting for augmenting web authentication: classification and analysis of methods," in *ACSAC*, 2016, pp. 289–301.

[21] N. A. Anagnostopoulos, T. Arul, Y. Fan, C. Hatzfeld, A. Schaller, W. Xiong, M. Jain, M. U. Saleem, J. Lotichius, S. Gabmeyer, J. Szefer, and S. Katzenbeisser, "Intrinsic run-time row hammer PUFs: Leveraging the row hammer effect for run-time cryptography and improved security," *Cryptography*, vol. 2, no. 3, p. 13, 2018.

[22] Z. Ba, S. Piao, X. Fu, D. Koutsonikolas, A. Mohaisen, and K. Ren, "ABC: enabling smartphone authentication with built-in camera," in *NDSS*, 2018.

[23] G. Baldini and I. Amerini, "Smartphones identification through the built-in microphones with convolutional neural network," *IEEE Access*, vol. 7, pp. 158 685–158 696, 2019.

[24] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre, "User tracking on the web via cross-browser fingerprinting," in *NordSec*, 2011, pp. 31–46.

[25] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, "Mobile device identification via sensor fingerprinting," *CoRR*, vol. abs/1408.1416, 2014. [Online]. Available: http://arxiv.org/abs/1408.1416

[26] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[27] E. Bursztein, A. Malyshev, T. Pietraszek, and K. Thomas, "Picasso: Lightweight device class fingerprinting for web clients," in *SPSM@CCS*, 2016, pp. 93–102. [Online]. Available: http://dl.acm.org/citation.cfm?id=2994467

[28] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via OS and hardware level features," in *NDSS*, 2017.

[29] A. Christensen, "Reduce resolution of performance.now," https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, 2015.

[30] W. B. Clarkson, "Breaking assumptions: Distinguishing. between seemingly identical items using cheap sensors," Ph.D. dissertation, Princeton, 2012.

[31] E. Commission, "General Data Protection Regulation (GDPR)," https://ec.europa.eu/info/law/law-topic/data-protection/eu-data-protection-rules_en.

[32] D. Cozzolino and L. Verdoliva, "Noiseprint: A CNN-based camera model fingerprint," *IEEE TIFS*, vol. 15, pp. 144–159, 2020.

[33] A. Das, G. Acar, N. Borisov, and A. Pradeep, "The web's sixth sense: A study of scripts accessing smartphone sensors," in *CCS*, 2018, pp. 1515–1532.

[34] A. Das and N. Borisov, "Poster: Fingerprinting smartphones through speaker," in *Poster at the IEEE Security and Privacy Symposium*, 2014.

[35] A. Das, N. Borisov, and M. Caesar, "Do you hear what I hear?: Fingerprinting smart devices through embedded acoustic components," in *CCS*, 2014, pp. 441–452.

[36] A. Das, N. Borisov, and E. Chou, "Every move you make: Exploring practical issues in smartphone motion sensor fingerprinting and countermeasures," *PoPETs*, vol. 2018, no. 1, pp. 88–108, 2018.

[37] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi, "Accelprint: Imperfections of accelerometers make

smartphones trackable," in *NDSS*, 2014.

[38] A. Durey, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "FP-Redemption: Studying browser fingerprinting adoption for the sake of web security," in *DIMVA*, 2021, pp. 237–257.

[39] P. Eckersley, "How unique is your web browser?" in *PETS*, 2010, pp. 1–18.

[40] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *CCS*, 2016, pp. 1388–1401.

[41] I. Fouad, N. Bielova, A. Legout, and N. Sarafijanovic-Djukic, "Missed by filter lists: Detecting unknown third-party trackers with invisible pixels," *PoPETs*, vol. 2020, no. 2, pp. 499–518, 2020.

[42] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *IEEE SP*, 2018, pp. 195–210.

[43] A. Gómez-Boix, P. Laperdrix, and B. Baudry, "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale," in *WWW*, 2018, pp. 309–318.

[44] C. Herder, M. M. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014.

[45] D. E. Holcomb, W. P. Burleson, and K. Fu, "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, no. 2, 2007, p. 01.

[46] ——, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Trans. Computers*, vol. 58, no. 9, pp. 1198–1210, 2009.

[47] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *IEEE SP*, 2021, pp. 283–301.

[48] H. Jonker, B. Krumnow, and G. Vlot, "Fingerprint surface-based detection of web bot detectors," in *ESORICS*, 2019, pp. 586–605.

[49] G. Kenneth Russell, personal communication.

[50] D. Kristol and L. Montulli, "HTTP state management mechanism," Internet Requests for Comments, RFC Editor, RFC 2109, Feb. 1997.

[51] P. Laperdrix, G. Avoine, B. Baudry, and N. Nikiforakis, "Morellian analysis for browsers: Making web authentication stronger with canvas fingerprinting," in *DIMVA*, 2019, pp. 43–66.

[52] P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints," in *IEEE SP*, 2016, pp. 878–894.

[53] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. V. Dijk, and S. Devadas, "A technique to build a secret key in integrated circuits with identification and authentication applications," in *In Proceedings of the IEEE VLSI Circuits Symposium*, 2004, pp. 176–179.

[54] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[55] M. Moenig, "Issue 820891: Webgl2: EXT_disjoint_timer_query_webgl2 failing in beta of 65," https://bugs.chromium.org/p/chromium/issues/detail?id=820891, 2018.

15

[56] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in JavaScript implementations," in *Proceedings of W2SP*, vol. 2, no. 11, 2011.

[57] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," *W2SP*, pp. 1–12, 2012.

[58] G. Nakibly, G. Shelef, and S. Yudilevich, "Hardware fingerprinting using HTML5," *CoRR*, vol. abs/1503.01408, 2015. [Online]. Available: http://arxiv.org/abs/1503.01408

[59] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *IEEE SP*, 2013, pp. 541–555.

[60] M. Perry, "Bug 1517: Reduce precision of time for JavaScript," https://gitweb.torproject.org/user/mikeperry/tor-browser.git/commit/?h=bug1517, 2015.

[61] S. Picek, "Challenges in deep learning-based profiled side-channel analysis," in *SPACE*, 2019, pp. 9–12.

[62] C. Project, "window.performance.now does not support sub-millisecond precision on Windows," https://bugs.chromium.org/p/chromium/issues/detail?id=158234#c110, 2016.

[63] T. Rokicki, C. Maurice, and P. Laperdrix, "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, Sep. 2021. [Online]. Available: https://hal.inria.fr/hal-03215569

[64] U. Rührmair, S. Devadas, and F. Koushanfar, "Security based on physical unclonability and disorder," in *Introduction to Hardware Security and Trust*. Springer, 2012, pp. 65–102.

[65] K. Russell, "Issue 859249: Extend WebGL 2.0 compute flag expiry to M85," https://chromium.googlesource.com/chromium/src.git/+/96186af9c385db253bf85f06f1324a729684cb2f, 2020.

[66] I. Sánchez-Rola, I. Santos, and D. Balzarotti, "Clock around the clock: Time-based device fingerprinting," in *CCS*, 2018, pp. 1502–1514.

[67] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *CVPR*, 2015, pp. 815–823.

[68] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Financial Cryptography and Data Security*, 2017, pp. 247–267.

[69] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *USENIX Security*, 2021.

[70] P. Snyder, C. B. Taylor, and C. Kanich, "Most websites don't need to vibrate: A cost-benefit approach to improving browser security," in *CCS*, 2017.

[71] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *DAC*, 2007, pp. 9–14.

[72] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-Scanner: The privacy implications of browser fingerprint inconsistencies," in *USENIX Security*, 2018, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/vastel

[73] ——, "FP-Stalker: tracking browser fingerprint evolutions," in *IEEE SP*, 2018, pp. 728–741.

[74] A. Vastel, W. Rudametkin, R. Rouvoy, and X. Blanc, "FP-Crawlers: Studying the resilience of browser fingerprinting to block crawlers," in *MADWeb*, 2020.

[75] J. von Kistowski, H. Block, J. Beckett, C. Spradling, K.-D. Lange, and S. Kounev, "Variations in cpu power consumption," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 147–158. [Online]. Available: https://doi.org/10.1145/2851553.2851567

[76] S. Wu, S. Li, Y. Cao, and N. Wang, "Rendered private: Making GLSL execution uniform to prevent WebGL-based browser fingerprinting," in *USENIX Security*, 2019, pp. 1645–1660.

[77] B. Zbarsky, "Clamp the resolution of performance.now() calls to 5us," https://hg.mozilla.org/integration/mozilla-inbound/rev/48ae8b5e62ab, 2015.

[78] J. Zhang, A. R. Beresford, and I. Sheret, "SensorID: Sensor calibration fingerprinting for smartphones," in *IEEE SP*, 2019, pp. 638–655.

[79] Z. Zhou, W. Diao, X. Liu, and K. Zhang, "Acoustic fingerprinting revisited: Generate stable device ID stealthily with inaudible sound," in *CCS*, 2014, pp. 429–440.

APPENDIX A
EVALUATION OF GEN 3, GEN 4, AND GEN 8 DEVICES

We report the complete evaluation for 600 traces, 20 points, and 5 iterations per point in the online setting, for the GEN 3 (Table IV), GEN 4 (Table V), and GEN 8 (Table VI) datasets.

TABLE IV. EVALUATION FOR THE GEN 3 DATASET, DEPENDING ON THE OPERATORS. THE BASELINE IS 10%

Operator	Accuracy	Median time (ms)
mul	93.5% ± 0.7%	3,097
sinh	89.5% ± 1.4%	8,757
abs	88.4% ± 1.0%	4,532
pow	87.5% ± 0.3%	4,663
log	87.1% ± 1.2%	9,839
exp2	87.0% ± 0.6%	7,532
shl	86.3% ± 1.7%	6,799
atanh	81.8% ± 1.4%	11,184
inversesqrt	80.1% ± 0.7%	6,799
trunc	67.1% ± 1.4%	1,667

TABLE V. EVALUATION FOR THE GEN 4 DATASET, DEPENDING ON THE OPERATORS. THE BASELINE IS 4%

Operator	Accuracy	Median time (ms)
mul	32.7% ± 0.3%	6,361
abs	29.3% ± 0.5%	3,295
shl	28.7% ± 0.4%	6,483
inversesqrt	28.2% ± 0.7%	6,485
exp2	27.8% ± 1.0%	6,528
trunc	26.6% ± 1.1%	3,161
log	25.3% ± 1.0%	7,673
pow	23.3% ± 0.6%	9,370
sinh	19.5% ± 0.5%	8,953
atanh	19.1% ± 0.6%	10,099

TABLE VI. EVALUATION FOR THE GEN 8 DATASET, DEPENDING ON THE OPERATORS. THE BASELINE IS 6%

Operator	Accuracy	Median time (ms)
exp2	$43.6\% \pm 1.2\%$	3,172
inversesqrt	$39.4\% \pm 0.9\%$	3,181
pow	$36.6\% \pm 1.4\%$	4,698
log	$33.6\% \pm 0.5\%$	3,299
sinh	$32.4\% \pm 0.9\%$	4,569
abs	$30.9\% \pm 0.6\%$	3,174
mul	$30.9\% \pm 1.0\%$	3,173
atanh	$30.6\% \pm 0.5\%$	5,935
trunc	$28.7\% \pm 0.6\%$	3,174
shl	$26.9\% \pm 1.1\%$	3,172

APPENDIX B
DETERMINISTIC ATTRIBUTES COLLECTED FOR THE IN-THE-WILD DATASET

```
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]
```

APPENDIX C
SELECTED HYPERPARAMETERS

Table VII summarizes the hyperparameters for the classifiers used in this work.

TABLE VII. HYPERPARAMETERS FOR THE CNN CLASSIFIER

Hyperparameter	Value	Space
Embedding size	256	32–256
Number of convolution blocks	3	1–10
Batch size	32	32–1024
Convolution filter size	128	8–128
Convolution kernel size	4	2–5
Dropout rate	0.119510	0–0.5
Activation	relu	relu, sigmoid

APPENDIX D
EVALUATION OF THE STANDALONE PIPELINE ON ADDITIONAL BROWSERS IN THE WILD

TABLE VIII. STANDALONE PERFORMANCE OF DRAWNAPART OVER MULTIPLE BROWSERS

Browser	Accuracy (Base rate)		
	Top-1	Top-5	Top-10
Chrome	24.31% (0.7%)	49.12% (2.9%)	60.80% (4.7%)
Edge	52.60% (2.9%)	85.48% (15.6%)	93.86% (29.7%)
Opera	79.28% (17.9%)	99.41% (50.7%)	100.0% (77.5%)
Yandex	89.69% (27.6%)	98.36% (85.9%)	99.76% (94.1%)

APPENDIX E
FP-STALKER HYBRID ALGORITHM WITH DRAWN APART ADDITION

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

1 **Function** *FingerprintMatching (F, f_u, λ, ϵ)*
2   **for** $f_k \in F$ **do**
3   **if** *FingerPrintHasDifferences(f_k, f_u, rules)* **then** $F_{ksub} \leftarrow$ exact $\cup < f_k >$;
4   **else**
5   exact $\leftarrow$ exact $\cup$ ¡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** $f_k \in F_{ksub}$ **do**
13   cosine_sim $\leftarrow$ GetSimilarity($f_u.avg\_embedding$, $f_k.avg\_embedding$);
14   **if** *cosine_sim* $> \epsilon$ **then**
15   **return** $f_k.id$
16   **end**
17     $< x_1, x_2, .., x_m >$ = FeatureVector(f_u, f_k);
18   p $\leftarrow$ P($f_u.id = f_k.id$ | $< x_1, x_2, .., x_m >$)
19   **if** $p \geq \lambda$ **then**
20   candidates $\leftarrow$ candidates $\cup < f_k, p >$
21   **end**
22   **end**
23   **if** $|candidates| > 0$ **then**
24   **if** $|GetRankAndFilter(candidates)| > 0$ **then** **return** *candidates[0].id* ;
25   **end**
26   **return** *GenerateNewId()*