Automatic Discovery of Emerging Browser Fingerprinting Techniques

Junhua Su
jsu6@ncsu.edu
North Carolina State University
USA

Alexandros Kapravelos
akaprav@ncsu.edu
North Carolina State University
USA

ABSTRACT

With the progression of modern browsers, online tracking has become the most concerning issue for preserving privacy on the web. As major browser vendors plan to or already ban third-party cookies, trackers have to shift towards browser fingerprinting by incorporating novel browser APIs into their tracking arsenal. Understanding how new browser APIs are abused in browser fingerprinting techniques is a significant step toward ensuring protection from online tracking.

In this paper, we propose a novel hybrid system, named BFAD, that automatically identifies previously unknown browser fingerprinting APIs in the wild. The system combines dynamic and static analysis to accurately reveal browser API usage and automatically infer browser fingerprinting behavior. Based on the observation that a browser fingerprint is constructed by pulling information from multiple APIs, we leverage dynamic analysis and a locality-based algorithm to discover all involved APIs and static analysis on the dataflow of fingerprinting information to accurately associate them together. Our system discovers 231 fingerprinting APIs in Alexa top 10K domains, starting with only 35 commonly known fingerprinting APIs and 17 data transmission APIs. Out of 231 APIs, 161 of them are not identified by state-of-the-art detection systems. Since our approach is fully automated, we repeat our experiments 11 months later and discover 18 new fingerprinting APIs that were not discovered in our previous experiment. We present with case studies the fingerprinting ability of a total of 249 detected APIs.

CCS CONCEPTS

• **Security and privacy → Privacy protections**.

KEYWORDS

Browser Fingerprinting, Web Measurement, Privacy, Program Analysis, Online Tracking

ACM Reference Format:
Junhua Su and Alexandros Kapravelos. 2023. Automatic Discovery of Emerging Browser Fingerprinting Techniques. In *Proceedings of Proceedings of the ACM Web Conference 2023 (WWW '23).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583333

1 INTRODUCTION

Web users browse the web from a plethora of devices and browser configurations. Although this makes the browsing experience more personal, it also exposes to visited pages enough information about the users' preferences to distinguish them. The monitor's resolution, the timezone, and even the GPU of the system (among several other configurations) can be combined to provide enough entropy to uniquely identify a browser. This stateless web tracking approach is known as *browser fingerprinting*.

Traditionally, tracking on the web has been conducted via stateful tracking techniques (e.g. third-party cookies). After decades of abuse, several modern browsers started taking measures to protect web users from stateful tracking by eliminating third-party cookies [20, 22, 25], disrupting this way one of the most profitable ecosystems in the world: web advertisements [26, 27]. Without traditional web tracking available, a major shift to alternative techniques is happening. Our work aims at disrupting the abuse of APIs for browser fingerprinting by continuously discovering emerging techniques and reporting them.

Since the discovery of browser fingerprinting, browser Application Programming Interfaces (APIs) [9] have played a central role in attack [38, 52, 56] and defense [29, 40, 51] mechanisms. Achieving a better understanding of the browser fingerprinting process in terms of APIs leads to building better tracking or anti-tracking tools. However, when a new browser feature is released, we do not have any system monitoring its abuse. State-of-the-art [30, 31, 49] in measuring browser fingerprinting behaviors in the wild has limitations: extra manual work is required to process the results that lack standard documentation, which is necessary for understanding the API. Besides, FingerprintAlert [30] does not work with JavaScript. FP-Inspector [49] detects fingerprinting behavior in script-level, and FP-RADAR [31] lacks dynamic analysis (§7.2).

In this paper, we develop an automated hybrid system based on VisibleV8 (VV8) [50], an instrumented Chromium browser that monitors the usage of browser APIs, and JStap [41], a static analysis tool that generates a dataflow graph of a given JavaScript file, to systematically discover browser APIs that are abused for browser fingerprinting in the wild over time. We rely on the key observation that browser fingerprinting techniques are commonly used together, which is shown by previous research [31, 36, 39, 49]. Based on this observation, we conceive and implement a data-driven algorithm that targets successively executed APIs around widely known fingerprinting APIs. We call the characteristics that APIs with the same purpose (e.g. fingerprinting) cluster together as locality.

The analysis required for our proposed methods is infeasible with the current web measurement tools. Therefore, we write a new VV8 post-processor that preserves the API execution sequence while the original post-processor destroys the sequence. We also conceive and implement static dataflow analysis of browser fingerprinting, which was not supported by JStap. To the best of our knowledge, we are the first to use static dataflow analysis on browser fingerprinting. Besides, we use a novel method based on character offset to combine static and dynamic analysis for better browser API detection used during browser fingerprinting.

As a result, our automated system discovers a total of 231 browser APIs (3.1% of total Chromium APIs) that are involved in browser fingerprinting by crawling the Alexa top 10k websites. We check every API detected by our system to determine if they are actively contributing to tracking or assisting the process. Out of the 231 fingerprinting APIs, 90 of them are direct and the remaining 141 are indirect. 11 months later, our system discovers 18 additional fingerprinting APIs in a new crawling experiment following the same procedure. This result indicates our system is able to monitor fingerprinting API abuse in the wild over time.

Our automated system overcomes limitations of existing fingerprinting behavior detection systems [30, 31, 49] and discovers 176 fingerprinting APIs that are not detected previously (§7.2). In summary, our main contributions are:

- We propose a novel automated hybrid system that is built based on VisibleV8 and JStap to identify previously-unknown fingerprinting APIs in the wild. The system can detect the abuse of newly developed APIs over time.
- We create an advanced locality-based algorithm in dynamic analysis and combine it with a unique static dataflow analysis to identify fingerprinting APIs.
- We detect a total of 249 fingerprinting APIs in two crawling experiments, and 176 of which are not detected by the state-of-the-art detection systems. We measure their usage in the wild (§6.2) and evaluate their fingerprinting ability (§6.4).

2 BACKGROUND

2.1 Browser Fingerprinting

Browser fingerprinting is a powerful technique that leverages unique browser characteristics and configurations to distinguish clients. Instead of using stateful tracking, like cookies, browser fingerprinting collects the discrepancies among clients' hardware, OS, and browser configuration to build a unique identifier for the user. To achieve this, trackers need to collect information from multiple sources in the browser by calling JavaScript APIs. All these small differences can be combined into a unique identifier which can replace stateful tracking in many tracking scenarios [11].

2.2 Browser APIs

Browser APIs are built inside the browser to help developers achieve advanced operations like retrieving browser or computing device information. Normally, standard documentation is published by major vendors like Mozilla for demonstrating syntactical usage and the concept of the browser API. WebIDL [18] is a collection of browser APIs under Chromium standards. On the contrary, **third-party APIs** are created by individuals or organizations. Although third-party APIs also serve to help developers build applications easier, their documentation is not published on the Web. Given a third-party API, it is hard to understand the usage of the API.

In the context of the web, APIs are built on top of the JavaScript language. Therefore, an API follows the format of `Object.function` or `Object.property`. In this paper, we use the word "**interface**" to refer to `Object`, "**feature**" to `function` or `property`, and "**API**" to the whole expression.

Direct Fingerprinting API. We define a browser API as a direct fingerprinting API if it returns more than one value when executing it on different computing environments. For example, `Navigator.language` returns the preferred language of a given browser. A website can directly use the return value of this API to distinguish between visiting browsers. We also restrict the return value that does not represent time (e.g. `Date.now`).

Indirect Fingerprinting API. Since the discovery of high-level fingerprinting techniques, direct fingerprinting APIs cannot cover all APIs used for fingerprinting. For example, in Canvas fingerprinting, many APIs are called to draw a graph. Then, the tracker uses a hash function on the graph to get a unique hash fingerprint. Without APIs used for drawing, the fingerprint cannot be generated but these APIs don't satisfy the definition of direct fingerprinting APIs. Therefore, we call them indirect fingerprinting API and we define an indirect fingerprinting API as an API that helps to generate a fingerprint but its return value cannot be directly used for fingerprinting. The union of direct and indirect fingerprinting APIs is fingerprinting APIs that can be used to generate a fingerprint.

Fingerprinting script. For brevity, we use the term "fingerprinting script" to describe a JavaScript file that contains many fingerprinting APIs. Determining the least amount of fingerprinting APIs that are required in a JavaScript file to uniquely fingerprint users is an open problem and out of scope for this work.

2.3 Program Analysis for Webpage Behavior

Static Analysis. Static analysis focuses on source code without executing the code. One common medium is the Abstract Syntax Tree (AST) which represents the syntactic usage of the given source code in the tree structure. Esprima [48] is the de facto tool to generate ASTs in research[41, 42, 61]. A Control Flow Graph (CFG) represents the logical expression of the given source code by demonstrating every possible execution state. A Dataflow Graph (DFG) represents the propagation of variables or values in a graph representation of a script.

Dynamic Analysis. Dynamic analysis is one type of analysis that requires executing the code while it can record the function usage or property access. In the context of JavaScript on the web, there are two main ways to monitor the JavaScript API calls, in-browser or in-band approach. In-band monitoring relies on the characteristics of the JavaScript language. Taking an example of the in-browser approach (e.g. VV8), Jueckstock *et al.*[50], modified the V8 engine which parses JavaScript inside chromium to collect native JavaScript function calls and property access.

3 CODE LOCALITY OF FINGERPRINTING TECHNIQUES

3.1 Intuition

Trackers today use fingerprinting APIs in a single script to uniquely identify users by executing them in a sequence and combining their outcomes together to craft a unique identifier. Based on this observation, we conceive a novel algorithm, called locality calculator, which allows our system to discover APIs used in proximity. Previous research [31, 36, 39, 49] shows that browser fingerprinting techniques appear together in fingerprinting scripts and no browser fingerprinting technique spans multiple files. However, APIs used in proximity do not necessarily contribute to fingerprinting. We take advantage of fingerprinting information in the dataflow graph to remove potential false positives (§4).

In fact, we never observed a single fingerprinting technique spanning multiple files in any of our experiments. Our approach generalizes and can be adapted in the future to accommodate multiple scripts. If trackers in the future split their fingerprinting functionality into multiple files, then we can stitch them back together based on the order of execution in the JavaScript engine.

3.2 Fingerprinting APIs Seed

To pinpoint a neighborhood of fingerprinting activity, we can preselect a list of well-known fingerprinting APIs that trackers use frequently (we call them *seed*), and locate their position in the sequence of API execution. We can discover previously unknown fingerprinting APIs by searching in the neighborhood of the seed for adjacent executed APIs.

One of the main goals of our paper is for our system to monitor the fingerprinting API abuse over time in the wild. Our design accepts configurable API seed, so it can provide up-to-date detection of related APIs and adapt to the evolution of stateless tracking techniques. For example, to discover which Canvas APIs are abused for fingerprinting in the wild, our system can be configured with Canvas APIs as seed and report APIs closely executed that will reveal all Canvas APIs abused in the wild. When fingerprinting techniques evolve in the wild, our system can capture the evolution by updating the seed. We envision browser vendors continuously running our system to track the evolution of fingerprinting.

3.3 Data-driven Per-Script Range for Locality

The key question is, within what range an API is considered as closely executed to the seed. We design the locality calculator to set different ranges based on different APIs executed by each website. If a large set of APIs is spotted from the seed and successively executed, the locality calculator will automatically mark the range larger and vice versa. The rationale is that if trackers use more fingerprinting APIs from the seed, this script is more likely to be fingerprinting users. Therefore, it is safer to expand the range of the neighborhood (and vice versa).

4 DATAFLOW OF FINGERPRINTING INFORMATION

Although fingerprinting APIs tend to cluster together, successively executed APIs do not necessarily contribute to fingerprinting. We leverage static dataflow analysis for fingerprinting to remove false positives. At a high level, after the tracker collects identifiable information from the victim's browser, the tracker creates a unique identifier for the user and sends it to the tracker's database. This creates a dataflow connecting pieces of information and ends with storing or transmitting the data. This dataflow is critical since browser fingerprinting is effective on a large number of users and the aggregation of user data is done through this dataflow. We label the part that stores or transmits data as *fingerprinting sink* and static fingerprinting dataflow analysis as *fingerprint analysis*. If the neighbor of the executed seed API does not contribute to fingerprinting, there should be no dataflow linking between the neighbor and a fingerprinting sink. Our fingerprint analysis can remove false positives.

We design our dataflow analysis aiming to detect connections between fingerprinting APIs that collect user data and fingerprinting sinks. These connections can be direct or indirect. A direct connection means that the return values of fingerprinting APIs are directly fed to the sink. An indirect connection means that the return values of fingerprinting APIs are processed (e.g. hashing or trimming) before reaching a sink. Due to the complexity of crawled scripts, we conduct the search **exhaustively**. In other words, dataflow analysis checks if the connection holds no matter how many intermediate steps are involved based on observations.

5 METHODOLOGY

Our framework, shown in Figure 1, consists of: (1) a crawler (§5.1), (2) a data collection and processing pipeline (§5.2), (3) a data analysis (§5.3) and produces a list of suspicious fingerprinting APIs (§6.1). The framework starts by processing the Alexa top 10k [14] domains into a VisibleV8 crawler. We develop a post-processor to extract useful data in the VisibleV8 logs. Then, we apply our locality calculator to find APIs executed closely to known fingerprinting APIs. After that, we annotate our logs with a dataflow graph produced by JStap and conduct a fingerprint analysis to verify the results of the locality calculator. Finally, our system produces a list of APIs that are suspicious for fingerprinting.

5.1 Crawling

We conduct experiments in a virtual machine running x86-64 Ubuntu 18.04.4 LTS (Bionic Beaver) running on an 8-core 2.20GHz Intel Xeon CPU. The crawler is built upon Puppeteer [13], a NodeJS-based [10] web automation framework that supports Chromium through Chrome DevTools Protocol [46]. We process the domain list by first concatenating the "http://" string followed by a domain from Alexa top 10k list [14] and using it as input to initiate our crawler written in JavaScript. If HTTP is not supported, the crawler will switch to an HTTPS connection. Then, we launch our instrumented Chromium (VisibleV8) with the default puppeteer option in headless mode and under a research university network. The crawler opens a browser tab and navigates to the concatenated URL. The default time for the tab to navigate one domain is 15 seconds. If there is ongoing data collection after 15 seconds, we allow an extra 30 seconds for navigating.

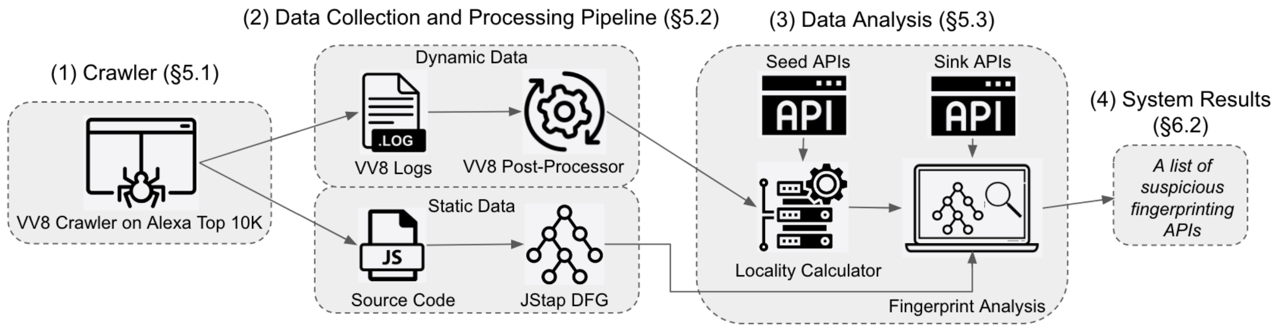

Figure 1: BFAD workflow consists of four main parts with: (1) Crawler (§5.1), (2) Data Collection and Processing Pipeline(§5.2), (3) Data Analysis (§5.3), and (4) System Results (§6).

5.2 Data Collection and Processing

5.2.1 Dynamic Browser API Calls. VisibleV8 (VV8) is an open-source instrumented version of Chromium that produces low-level JavaScript behavior logs by visiting a webpage [50]. The logs contain all executed APIs and their context: origin names of JavaScript executed in the crawled domain, all executed JavaScript source code (including dynamic code), and all browser APIs that were invoked during crawling. VV8 also provides the location in the JavaScript source code for each executed API which allows us to map a dynamically monitored API call back to its source code. Visiting a domain, our system collects scripts through Chrome DevTools Protocol, specifically the debugger [47]. Meanwhile, we get the VV8 logs and use our post-processor on VV8 logs to generate a sequence of executed APIs with their source code offsets (e.g. index location in the script).

VisibleV8 modifications. VisibleV8 logs (one example can be found here) include all executed browser APIs in their order of execution but are too complicated to read or directly process. Thus, a post-processor is required to extract useful information from the logs. However, the original VisibleV8 post-processor only counts the frequency of APIs and omits code locality. To enable the locality calculator that depends on the code execution sequence, we create a new VisibleV8 post-processor to capture this key relationship. We make our modifications to VisibleV8 publicly available (§I), so that other researchers can leverage code locality algorithms for web measurements.

5.2.2 Static Dataflow Graph. We use a dataflow graph to remove false positives by analyzing whether an API has a dataflow to a sink. For our dataflow graph generation, we rely on JStap [41], which is a tool used for generating a Program Dependency Graph (PDG) for a given JS file and it is built on top of Esprima [48]. According to their implementation, a PDG is a combination of an abstract syntax tree (AST), control-flow graph (CFG), and dataflow graph (DFG). In our system, we directly feed crawled JavaScript files to JStap to generate their DFGs and concentrate on dataflow dependency.

JStap Modifications. Although we use JStap to generate dataflow graphs, JStap has no support for browser fingerprinting. We expand JStap by introducing dataflow analysis logic for fingerprinting described in Section 4 including the dataflow sinks in Section 5.3.2. We conceive an approach by using the source code location of executed APIs (§5.3.4) to combine VisibleV8 and JStap. We can use fingerprint analysis to verify the results of dynamic analysis for fewer false positives. All these modifications are necessary for detecting closely executed fingerprinting APIs in the wild and no previous research considered static fingerprint analysis.

5.3 Data Analysis

5.3.1 Seed Fingerprinting APIs. The locality calculator is a generic algorithm that returns closely executed APIs with respect to given APIs. To make the locality calculator aware of browser fingerprinting, we leverage a list of APIs that are well-known to be abused for browser fingerprinting. We refer to previous research [29, 39, 45, 52, 53, 57, 65] that includes well-known APIs which can assist in browser fingerprinting. Based on previous research, we select 35 fingerprinting APIs.

Our seed list consists of three APIs from `CanvasRenderingContext2D`, one from `WebGLRenderingContext`, three from `WebGL2RenderingContext`, five from four audio-related interfaces, 15 from `Navigator`, five from `Screen`, one from `Window`, and two from `Geolocation`. We will call this list a "seed" list in the paper.

5.3.2 Dataflow Sink APIs. We summarize a total of 17 browser APIs that can be treated as a sink. These APIs are either used for data transmission (e.g. `WebSocket.send`) or data storage (e.g. `Window.localStorage`). We also refer to previous research [42, 54, 62] on forming our sink APIs list. A full list of seed and sink APIs is listed in Github page.

5.3.3 Locality Calculator. After crawling, collecting data, and processing the data, we will get an array of executed APIs in sequence. Then, we feed a sequence of executed APIs (with their offsets) to the locality calculator. Assume this sequence contains a subsequence of successively executed seed APIs and this subsequence has n APIs. The number of executed APIs between a neighbor API and the closest API in this subsequence is m. Our algorithm assigns this neighbor API an integer weight equal to n minus m if n is larger than m. Otherwise, our algorithm assigns zero. If the neighbor API is close to more than one subsequence, repeat the previous assignment to each subsequence and assign this neighbor API a sum of weight from each subsequence. Then, we consider a neighbor API with positive weight as a closely executed API and feed it (with its

Table 1: Crawling statistics on Alexa top 10k

Domain	8,446
Origin	11,709
Script	474,659
Total API Calls	753,998,537
Total API Calls without duplication	235,771
Browser API Calls	218,143,537
Browser API Calls without duplication	2,333

character offset) to fingerprint analysis. We list the pseudo-code in Appendix 1.

5.3.4 Fingerprint Analysis. VV8 can monitor the execution of an API and its location in the source code. Via string indexing, we can identify the location of this executed API in the source code. This location is called a character offset. In the DFG generated by JStap, every node has a range. Indexing this range in the source code can also identify the location of a node, which is equivalent to an API in the source code. If we run a script with VV8 and JStap, VV8 outputs character offsets of a list of executed APIs, and JStap outputs character offsets of nodes that correspond to these APIs. In this script, character offsets generated by JStap are the same as character offsets generated by VV8. Therefore, having the list of executed APIs and respective character offsets (the character offsets in VV8 will not be changed when input to the locality calculator), we can index them in the source code. We also use JStap to create a DFG based on the same source code. In this DFG, a set of nodes have the same character offsets. According to the same character offsets, we can map out executed APIs in the nodes in DFG. With this mechanism, we use fingerprint analysis to ensure the results of the locality calculator connect to a sink for fingerprinting purposes. The logic of fingerprint analysis is stated in Section 4.

6 RESULTS

6.1 Crawling Results

Following the approach described in the methodology section, we introduce the basic statistics of the crawling process in the beginning. The first crawling of Alexa top 10k was completed in October 2021. Table 1 represents the overall statistics of the collected data. The left column states the types of crawled data and the right column states the quantity of each type. From top to down, the first row shows how many domains are successfully crawled among Alexa top 10k. There are many reasons that cause crawling to fail, including domain name not resolved, SSL protocol error, and connection refusal. We successfully crawl 8,446 websites. We observed 11,709 origins loading 474,659 scripts. This is 56 and 46 times the number of domains and origins respectively and indicates the high complexity of the web. The total number of observed function calls is over 700M, out of which 200k were unique function calls executed. We cross-check the function calls with the WebIDL file [18] specification and isolate over 200M browser API calls. Almost 30% of the function calls are browser APIs. As said previously, we focus on browser APIs because they are supported by standard documentation from major vendors. Removing duplicate calls on browser API calls, 2,333 browser APIs are executed, which is 31.3% of all browser APIs (7,447) that exist in the WebIDL specification. The total number of executed function calls (third-party API plus browser API) is 100 times more than the number of executed browser APIs.

Table 2: Composition of verified results

Fingerprinting APIs	Direct APIs	90
	Indirect APIs	141
Assisting APIs	URL-related APIs	29
	Sink APIs	33
	(De)Obfuscation	4

Overall, we discover 297 browser APIs participating in fingerprinting techniques: 231 fingerprinting APIs and 66 APIs that assist the fingerprinting process, like obfuscation or additional sinks. The composition of these APIs is listed in Table 2. The locality calculator discovers 369 browser APIs and the fingerprint analysis verifies that 323 out of 369 browser APIs have a dataflow that contains a sink.

We analyze that 46 APIs do not have a dataflow to a sink and discover that five of them are used for the DOM and HTML setup, 12 of them are sink APIs, 18 APIs are fingerprinting APIs, and the rest cannot be used for fingerprinting. These 18 APIs are not included final result (297 APIs) and our static system rejects these 18 APIs that can be used for fingerprinting because 1) the source file takes more than 5 minutes to generate DFG, 2) the source code contains an unexpected token and DFG cannot be generated (esprima parse error), and 3) some APIs are used for the legit purpose in the crawled case.

Results Verification. We verify that 297 out of 323 APIs are either fingerprinting APIs or assisting APIs. In 26 false positives, two APIs are from `AbortControler` and one is `UnderlyingSourceBase.type`. We cannot find a connection between them and fingerprinting. The rest of the false positives are the DOM or HTML-related APIs. They are selected by our system because the webpage is a mix of the DOM, HTML, and JavaScript. Discovering fingerprinting APIs in the wild by using proximity inevitably introduce the usage of the DOM or HTML-related APIs. Verification details are in Appendix B.

Fingerprinting APIs. In Table 2, we classify our results into categories based on definitions in Section 2.2 with explicit steps listed in Appendix B. 90 (39.0%) out of 231 fingerprinting APIs are direct fingerprinting APIs. Direct fingerprinting APIs are not the majority because they only demonstrate a fraction of simple fingerprinting techniques. The majority of direct ones are related to browser information. Graphic-related and performance-related APIs are indirect as they require a relatively complicated setup.

Assisting APIs. Assisting APIs are used to transmit users' information or obfuscate fingerprinting scripts but do not reveal more entropy to trackers. In our analysis, we discovered additional sink APIs. As stated in Section 4, sink APIs are essential for trackers, and because new browser features can also act as sinks, we cannot include all of them in our original sink list that we start with, but our system can identify them. The number of sink APIs found by our system is 33 in total. Another category is URL-related APIs. This category includes interfaces like `HTMLAnchorElement`, `Location`, and `URL`. They are critical for data transmission, especially to

Table 3: Usage of newly discovered fingerprinting APIs

# APIs	5	10	15	20	25	30	35
# Scripts	10,224	5,402	2,826	1,387	738	444	257
Percent	81.6%	43.1%	22.6%	11.1%	5.6%	3.5%	2.1%

third-party trackers. When fingerprinting scripts are third-party, trackers need to know the URL of the first-party domain that includes third-party scripts. Otherwise, the trackers do not know what content the user is viewing and collected information becomes less valuable. After the trackers collect information, they need to send data externally. For trackers who do not reveal the source code, they obfuscate the source by using browser APIs like `TextEncoder.encode` and `window.atob`. Our system also identifies four APIs used for (de)obfuscation.

We share the full list of our verified results, 323 browser APIs, in an Github page. For fingerprinting APIs, we also list the documentation from the browser vendor (e.g. Mozilla), a functionality demonstration, and whether they are deployed on our website along with the API name. We hope this extensive list of fingerprinting APIs will help other researchers and browser vendors.

6.2 Usage of Discovered Fingerprinting APIs

We conduct two experiments to show how these newly discovered fingerprinting APIs are abused in the wild in terms of numerical data. The first experiment is to measure how many scripts are using specific numbers of newly discovered fingerprinting APIs. 12,531 scripts have executed at least one seed and one sink API. In Table 3, there are 10,224 scripts (out of 12,531) that executed more than five discovered APIs. When the threshold increases, the number of scripts decreases which makes sense. We argue that scripts which execute more newly discovered APIs are more likely to be fingerprinting scripts. With the manual inspection of the top 50 scripts with the most numbers of newly discovered fingerprinting APIs usage, we find all of them are fingerprinting scripts.

The second experiment measures the usage of newly discovered fingerprinting APIs in fingerprinting scripts. Since our work does not include finding fingerprinting scripts, we use fingerprinting domains generated by FP-Inspector [24] and disconnect [37]. We take the union of these two lists of fingerprinting domains and take the intersection with our crawled domains. We find out there are 792 fingerprinting domains in our crawled data. Among them, the average number of executed newly discovered fingerprinting APIs is 11.5. Besides, we cross-compare them with two well-known fingerprinting libraries and find out 22 APIs used in fingerprintjs2 [23] and 18 APIs used in cross-browser fingerprinting [21].

6.3 System Usability Over Time

To demonstrate that our system can be reused over time to continuously monitor the abuse of fingerprinting APIs, we redid the crawling on the same 10K domains in September 2022 (11-month gap) and applied our system to this data set by following the same procedure. As result, the locality calculator identifies 374 suspicious APIs and fingerprint analysis confirms 304 APIs. Cross-checking with the previous run, our system identifies 41 new APIs which are not included in Table 2. After analyzing them, 18 of them are fingerprinting APIs, 16 of them are sink APIs, and the rest 7 APIs are the DOM-related. Among 18 fingerprinting APIs, 15 of them are not detected by state-of-the-art detection systems. We list 18 fingerprinting APIs in Github page.

6.4 Evaluation of Discovered Fingerprinting APIs

We evaluate fingerprinting capabilities of APIs (including APIs found in the second crawling) not detected by state-of-the-art in the following paragraphs. Due to limited space, we explain discovered APIs under popular fingerprinting interfaces (e.g. Navigator, Canvas, WebGL(2), and Performance) in Appendix C.

To better demonstrate the fingerprinting techniques abused in the wild, we provide the scripts that execute detected fingerprinting APIs. We verify that all of them are fingerprinting scripts and state-of-the-art detection systems can not detect any of them except for the ebay script. Furthermore, these fingerprinting scripts are widely deployed to fingerprint a huge population of users. These scripts are from Google ads service, Youtube, Microsoft, eBay, and other popular domains which have a huge number of visits every day. Beyond that, our system also discovers suspicious fingerprinting scripts but we cannot manually analyze them due to obfuscation.

Window. The `Window` interface serves as a global object to represent the browser window. Our system newly discovers 41 APIs from this interface. Among them, we notice that there are pairs of Window features that return the same results, for example, Window.clientInformation-Window.navigator and Window.screenY-Window.screenTop respectively. If the defense mechanism modifies the return value from one feature but not another, it saves the tracker's time by making itself unique.

Another category is related to user interaction, `EventHandler` [6]. The mechanism of this type is to collect users' activities during their visiting period, like `Window.ontouchstart` for user's interaction with the touch surface. This category can be effectively used to distinguish between bots and humans, and record users' behavior through a mouse, touch surface, or keyboard that provides ample side-channel information.

The window object also contains APIs to return whether the status bar or location bar of the browser is visible. Users can hide these bars based on configuration options and trackers are taking advantage of this. Real-world exploitation can be found from sardine with another example in akamaized. Note that major browser vendors have patched it to only return true for privacy concerns [19] while our system catches fingerprinting scripts still abusing it after the patch.

MutationObserver-related. `MutationObserver` and `MutationRecord` help keep track of changes made to the DOM and our system newly discovers five APIs. The trackers can use `MutationObserver` to monitor the DOM. If users adopt defending extension which is based on modifying the DOM tree, the trackers know about it and users will become unique in this case. This technique is demonstrated by Vastel *et al.* [66] and we also observe its abuses in the wild. Taking deeper inspection, we find Youtube and Microsoft scripts incorporate `MutationObserver` APIs to collect users' behavior.

IntersectionObserver. Also, we are the first to observe abuses of six APIs under `IntersectionObserver` in the wild and they provide a way to measure what percentage of a given element (e.g. video pop-ups) is visible to the user's viewpoint. By using this interface, trackers can know: 1) whether the content on the web page appears on users' screens, and 2) how long users stay on a given element. Then, trackers can distinguish humans from bots by calculating the viewing speed since bots have incredible scrolling speed and precise staying time. For example, `isIntersecting` tells whether the given element is inside the users' screen, and `intersectionRatio` tells the percentage of the given element is inside the users' screen. We find they are abused in cloudfront and moatads fingerprinting scripts.

6.5 Case Studies

We demonstrate several discovered fingerprinting APIs in the following case studies. We analyze them based on their documentation and code snippets (listed in the Appendix D) from crawled source code. These APIs are closely executed by known fingerprinting APIs, have a connection to sink, and are not discovered by state-of-the-art. Due to limited space, we pack them with some portion omitted, and the well-formatted version with more context is provided in the Github page.

History.length returns the number of the web page visited by a given tab from a user. It is a good indicator for the website to know if the user directly navigates the domain or visits the domain from other domains. We observe that they are used by google ad service conversion and async version scripts.

ServiceWorkerContainer.controller is designed to provide users' offline browsing experience and run in the background along with other service worker APIs. We observe that abuse of this API from a Russian news website and wikiland scripts. This API can be abused for fingerprinting whether service work APIs are supported or controlled with a given browser. Moreover, Performance timing APIs can be applied to measure the start timing of workers.

MediaQueryList.matches can be used to monitor the CSS element with respect to a provided media query, like browser window size. In provided fingerprinting scripts, it is used to detect whether a given browser is from a smartphone by checking the screen size. Combining with `MediaQueryList.addListener`, the change of media query can also be captured for fingerprinting purposes. Our system detects taboola and cheqzone are using it.

PageTransitionEvent.persisted tells whether a webpage is loaded from cache. Similar to History.length, it can show whether a user visited a given webpage before. From Kugou and amazon aws scripts, it is used with `Window.addEventListener` and `pageshow` event for fingerprinting.

Permissions APIs ask users' consent before enabling corresponding functionalities (e.g. camera). In ebay and securedtouch scripts, we find that tackers can iterate all permissions and collect their states. Users with different permission settings are distinguished by this technique.

7 RELATED WORK

7.1 Development of Browser Fingerprinting

In 2010, Eckersley [38] from the Electronic Frontier Foundation collected 470,161 fingerprints by broadcasting on social media and popular websites. He quantified "quirkiness" with an information theory-based interpretation, entropy, and demonstrated that 18.1 bits of entropy help distinctively identify 286,777 browsers in the best-case scenario.

Since then, the enrichment of browser features has prompted a line of research on finding new attributes of browser fingerprinting. For example, Mowery *et al.* [56] discovered a high entropy method with WebGL [17] and HTML Canvas [7]. Cao *et al.* [34] demonstrate the ability of cross-browser fingerprinting based on Canvas and WebGL. Laperdrix *et al.* [52] explored the validity of browser fingerprinting with 17 attributes and over 100,000 fingerprints. Meanwhile, they proved how the fingerprinting technique is applied to mobile phones. Das *et al.* [36] found four types of smartphone sensors that contribute to the fingerprint. System font list [43], browser extensions [64], evercookie [28], Battery API [58], and the Audio API [39] are also marked as entropy source by corresponding researchers.

7.2 Comparison with the state-of-the-art

To the best of our knowledge, Al-Fannah *et al.* [30], Bahrami *et al.* [31], and Iqbal *et al.* [49] also discovered fingerprinting features by crawling the web. We refer to them as state-of-the-art in this paper and individually compare our work with them in the following.

FP-Inspector. Iqbal *et al.* [49] built a machine learning-based fingerprinting script detector named FP-Inspector and they adopted the functionality of OpenWPM [39] to crawl Alexa top 100k websites. Their work focuses on detecting browser fingerprinting scripts and not on the underlying browser APIs responsible for the problem. Studying browser fingerprinting at a granular API level leads to a better understanding of fingerprinting techniques and allows researchers directly modify these APIs to protect users.

They discovered 161 JavaScript keywords that executed much more in fingerprinting scripts than non-fingerprinting ones. Unlike JavaScript keywords, our system yields result in browser API. With browser API, there is no need to spend the extra manual effort to filter out keywords that do not have documentation or correspond to multiple APIs. Cross-checking their results with the WebIDL file and excluding interface keywords, 84 (52.1%) features are browser APIs. This low standard feature percentage tells that almost half of their results may be infeasible to analyze. Due to the choice of measurement tool, their system can only measure the usage of a pre-selected list of APIs that they know beforehand. On the other hand, our system monitors every native JS API execution. Their definition of fingerprinting considers Canvas, WebGL, font, and Audio without considering other techniques (e.g. under Window and Navigator). Our broader definition leads to a more comprehensive fingerprinting behavior detection. The other two papers do not have a clear definition of fingerprinting.

Among these 84 standard features, our system can detect 34 (40.4%) of them. For 50 features our system does not cover, 16 (32%) features are executed no more than two times during our crawling process. After inspection, the major reason why we have this low coverage is the insufficient amount of seed. Therefore, by adding fingerprinting APIs discovered by our system to the seed (details in Appendix E), the coverage is 81% while the remaining features are barely executed during crawling.

FP-RADAR. With the concept of "guilt by association", Bahrami *et al.* [31] built a system to detect fingerprinting features based on 10-year longitudinal data they crawled from Alexa top 100k websites. Although we have a similar assumption, we have distinct interpretations of the assumption and system architectures. Their system only contains static analysis while our hybrid system is better at dealing with obfuscation and minification which hinder static analysis. They took a machine-learning approach while we rely on our new locality algorithm. Also, our system provides a broader picture of browser fingerprinting by taking dataflow into consideration when no previous research on fingerprinting uses static dataflow analysis.

Their results are 313 features from a cluster that share the highest similarity with known fingerprinting scripts identified by FP-Inspector. They publicized 50 APIs (out of 313) with manually labeled interfaces while the rest features are not publicized. Speaking of the coverage, 33 (66%) discovered features out of 50 public ones have corresponding browser APIs. Among 33 features, 11 are executed no more than two times during crawling. Our system covers 17(51%) with original seed and 21(63.6%) with extended seed.

FingerprintAlert. Al-Fannah *et al.* [30], constructed a crawler and collected transfer data (e.g. HTTP response message) to manually find fingerprinting attributes. However, they collected transmitted data without involving JavaScript. Since transmitted data is totally controlled by the sender, their method is also suffered from code transformation techniques. It turns out that 65 out of 286 (22.7%) of their finding attributes correspond to browser APIs. This low browser API ratio is due to a lack of JavaScript-level monitoring. Moreover, there is no definite relationship between their collected data and specific browser API. It makes their results provide little information for further analysis. Our system, including our seed and sink lists, covers 57 (87.7%) of them.

Summary. Our system has several advantages compared to the state-of-the-art: 1) it supports long-time monitoring with customizable API seeds, 2) produces results as browser APIs that do not require manual filtering, 3) has a broader coverage of fingerprinting techniques, 4) leverages static dataflow analysis, and 5) bypasses code transformation techniques. Meanwhile, by adding fingerprinting APIs to the seed, our system can cover the vast majority of features discovered by state-of-the-art if we can observe ample API calls. Removing APIs containing attributes in the state-of-the-art, we discover a total of 161 out of 231 fingerprinting browser APIs that have not been reported by state-of-the-art.

7.3 Program Analysis

Englehardt *et al.* [39] adopted OpenWPM on Alexa top one million websites to monitor several fingerprinting patterns(e.g. AudioContext [16], Canvas [7], Battery [2]). A wide list of research [32, 36, 49, 55, 60] either chose it or modified it as their tool due to its reliability. In terms of the in-band approach, Snyder *et al.* [63], overwrite JavaScript function calls and use `Object.watch` to measure usage of function calls and property access respectively.

Due to the low computational cost, static analysis is often used for malicious code detection with its inherent advantage of accessibility. Fass *et al.* built JStap [41], a modular system that can generate AST, CFG, and DFG. Canali *et al.* [33] utilized the features of HTML and JavaScript code as a filter to efficiently reduce the number of suspicious web pages, leaving only a few for costly dynamic and manual analysis. Curtsinger *et al.* [35] used hierarchical features of the JavaScript AST to identify malware with Bayesian classification. Rieck *et al.* [59] proposed static and dynamic detection models as an extension to a web proxy for the analysis of malicious patterns.

8 LIMITATIONS & DISCUSSION

Although many of the discovered APIs are known to the community, these APIs are scattered in multiple reports and are often discovered manually without knowing how they are abused in the wild. Our system provides a novel systematic way of discovering fingerprinting APIs abused in the wild collectively and can be deployed continuously to discover new APIs, something that no previous work can achieve. Our system discovers 24 fingerprinting APIs that are not known to the community to the best of our knowledge. Since our system focuses on fingerprinting behavior detection in the wild, we select systems with a similar approach (i.e. crawling-based) as state-of-the-art rather than the state-of-the-art fingerprinting scripts (e.g. FingerprintJS) for a valid comparison.

Speaking of ways to evade our system, trackers can purposefully distribute fingerprinting APIs execution, it is easy for our system to capture it by count the number of executed fingerprinting APIs. It is another advantage of API-level fingerprinting analysis since trackers cannot fingerprinting without executing fingerprinting APIs.

9 CONCLUSION

In this paper, we proposed a hybrid system named BFAD that can identify previously unknown fingerprinting APIs by locating known fingerprinting APIs in an executed sequence of APIs and searching for their neighbors. By building on top of VisibleV8 and JStap, BFAD employs a novel locality algorithm and a fingerprint analysis. Combining them together with character offset, BFAD discovers a total of 249 fingerprinting APIs starting from 35 fingerprinting APIs and 17 sink APIs with two crawling experiments. The second crawling that happens 11 months later gives us 18 new fingerprinting APIs. Compared to state-of-the-art, our automated system detect 176 (out of 249) fingerprinting APIs that are not detected by state-of-the-art. We also evaluate fingerprinting ability of these APIs and show how they are abused. We envision browser vendors leveraging our system in order to evaluate the risks of released browser features and monitor their abuse over time (i.e. Privacy Budget [8]). We also make the crawled data and source code available.

10 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments and suggestions on how to improve this paper. We also thank Lichen Fu for his help on this work. This work was supported by the National Science Foundation (NSF) under grants CNS-2138138 and CNS-2047260.

REFERENCES

[1] 2021. AmIUnique. https://amiunique.org. (2021).
[2] 2021. Battery Status API. https://www.w3.org/TR/battery-status/. (2021).
[3] 2021. BrowserLeaks - Web Browser Fingerprinting - Browsing Privacy. https://browserleaks.com. (2021).
[4] 2021. Device Info. https://www.deviceinfo.me/. (2021).
[5] 2021. Fingerprinting JSEcho. http://privacycheck.sec.lrz.de/active/fp_je/fp_js_echo.html. (2021).
[6] 2021. GlobalEventHandlers. https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers. (2021).
[7] 2021. HTML Canvas 2D Context. https://www.w3.org/TR/2dcontext/. (2021).
[8] 2021. Introducing the Privacy Budget. https://www.youtube.com/watch?v=0STgfjSA6T8&ab_channel=GoogleChromeDevelopers. (2021).
[9] 2021. JavaScript APIs. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API. (2021).
[10] 2021. node.js. https://nodejs.org/en/. (2021).
[11] 2021. Panopticlick. https://panopticlick.eff.org. (2021).
[12] 2021. Pixelscan. https://pixelscan.net/. (2021).
[13] 2021. Puppeteer. https://pptr.dev/. (2021).
[14] 2021. top-1m. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip. (2021).
[15] 2021. UNIQUEMACHINE. http://uniquemachine.org/. (2021).
[16] 2021. Web Audio API. https://www.w3.org/TR/webaudio/. (2021).
[17] 2021. WebGL: 2D and 3D graphics for the web. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. (2021).
[18] 2021. WebIDL. https://www.w3.org/TR/WebIDL-1/. (2021).
[19] 2022. BarProp.visible. https://developer.mozilla.org/en-US/docs/Web/API/BarProp/visible. (2022).
[20] 2022. Building a more private web: A path towards making third party cookies obsolete. https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html. (2022).
[21] 2022. cross_browser. https://github.com/Song-Li/cross_browser. (2022).
[22] 2022. Disable third-party cookies in Firefox to stop some types of tracking by advertisers. https://support.mozilla.org/en-US/kb/disable-third-party-cookies?redirect=no. (2022).
[23] 2022. fingerprintjs. https://github.com/fingerprintjs/fingerprintjs/tree/v2. (2022).
[24] 2022. FP-Inspector. https://github.com/uiowa-irl/FP-Inspector/blob/master/Data/fingerprinting_domains.json. (2022).
[25] 2022. Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/. (2022).
[26] 2022. Internet Advertising Revenue Report: Full Year 2021. https://www.iab.com/insights/internet-advertising-revenue-report-full-year-2021/. (2022).
[27] 2022. Online advertising revenue in the United States from 2000 to 2021. https://www.statista.com/statistics/183816/us-online-advertising-revenue-since-2000/. (2022).
[28] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[29] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: Dusting the Web for Fingerprinters. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[30] Nasser Mohammed Al-Fannah, Wanpeng Li, and Chris J Mitchell. 2018. Beyond cookie monster amnesia: Real world persistent online tracking. In *International Conference on Information Security*.
[31] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal measurement and early detection of browser fingerprinting. *Proceedings on Privacy Enhancing Technologies* (2022).
[32] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. 2021. Reining in the Web's Inconsistencies with Site Policy. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.
[33] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. 2011. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the International World Wide Web Conference (WWW)*.
[34] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*.
[35] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the USENIX Security Symposium*.
[36] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. 2018. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[37] Disconnect. 2021. disconnect-tracking-protection. https://github.com/disconnectme/disconnect-tracking-protection. (2021).
[38] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*.
[39] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-Million-Site Measurement and Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[40] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. 2015. FPGuard: Detection and Prevention of Browser Fingerprinting. In *Data and Applications Security and Privacy XXIX*.
[41] Aurore Fass, Michael Backes, and Ben Stock. 2019. JStap: A Static Pre-Filter for Malicious JavaScript Detection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.
[42] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[43] David Fifield and Serge Egelman. 2015. Fingerprinting web users through font metrics. In *International Conference on Financial Cryptography and Data Security*.
[44] Henrik Gemal. 2021. BrowserSpy.dk. http://browserspy.dk/. (2021).
[45] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the Crowd: An Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *Proceedings of the 2018 World Wide Web Conference*.
[46] Google Chrome. 2021. https://chromedevtools.github.io/devtools-protocol/. (2021).
[47] Google Chrome. 2021. https://chromedevtools.github.io/devtools-protocol/tot/Debugger/. (2021).
[48] Ariya Hidayat. 2021. ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/. (2021).
[49] U. Iqbal, S. Englehardt, and Z. Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[50] Jordan Jueckstock and Alexandros Kapravelos. 2019. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*.
[51] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. 2017. FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *ESSoS 2017 - 9th International Symposium on Engineering Secure Software and Systems*.
[52] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[53] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. 2016. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *Proceedings of the USENIX Security Symposium*.
[54] Tianyi Li, Xiaofeng Zheng, Kaiwen Shen, and Xinhui Han. 2021. Poster: FPFlow: Detect and Prevent Browser Fingerprinting with Dynamic Taint Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[55] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. 2017. Measuring the Insecurity of Mobile Deep Links of Android. In *Proceedings of the USENIX Security Symposium*.
[56] Keaton Mowery and Hovav Shacham. 2012. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Proceedings of W2SP 2012*.
[57] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Chris Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[58] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. 2016. The Leaking Battery. In *Data Privacy Management, and Security Assurance*.
[59] Konrad Rieck, Tammo Krueger, and Andreas Dewald. 2010. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.
[60] Valentino Rizzo, Stefano Traverso, and Marco Mellia. 2020. Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning. *Proceedings on Privacy Enhancing Technologies* (2020).
[61] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*.
[62] Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld. 2021. Essentialfp: Exposing the essence of browser fingerprinting. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.
[63] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser Feature Usage on the Modern Web. In *Proceedings of the 2016 Internet Measurement Conference*.
[64] Oleksii Starov and Nick Nikiforakis. 2017. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[65] Oleksii Starov and Nick Nikiforakis. 2018. PrivacyMeter: Designing and Developing a Privacy-Preserving Browser Extension. In *Engineering Secure Software and Systems*.

[66] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *Proceedings of the USENIX Security Symposium.*

A APPENDIX

B EXPLICIT STEPS ON FINGERPRINTING APIS VERIFICATION

Generally, we first check if this API is demonstrated in fingerprinting demonstration websites [1, 3–5, 11, 12, 15, 44] or previous research. If this API is demonstrated before and satisfies our definition, we mark it as a fingerprinting API. Based on the documentation, we also implement this API on our website and test it with devices controlled by us. If we can get users' information by directly using the return value of this API, we mark it as a direct fingerprinting API. Otherwise, we mark it as an indirect fingerprinting API. If the API doesn't follow the above steps, we thoroughly review the JS source code that executes the API. If the JS source file contains many known fingerprinting APIs in the seed list and the API satisfies our definition, we still mark it as a fingerprinting API.

C DISCOVERED FINGERPRINTING INTERFACES

Navigator. `Navigator` interface contains information about the browser. Because it's easy to access and usually contains distinguishable browser information, it's a profitable target for trackers. Our system newly discovers 7 APIs under the Navigator interface. Among the APIs our system identifies, there are two types of navigator features: hardware and browser information. The hardware-related features include `Navigator.xr` and `Navigator.keyboard`. They help trackers know whether a VR device or keyboard is connected to the visiting computer. Furthermore, trackers can gain detailed information about the connected deceives if permission is allowed.

For the rest, `Navigator.userAgentData` can be used to obtain platform from the user agent, device type (e.g. mobile), and browser information while `navigator.appVersion` provides similar information with more details. `Navigator.connection` yields `Network-information` object which reveals effective type of the connection(e.g. 2g, 3g, and 4g) and saveData option.

Canvas and WebGL. Graphic processing-based fingerprinting originates from different image rendering pipelines provided by different graphic cards and operating systems [34, 56].

State-of-the-art discovers features mostly from WebGL(2) and misses Canvas-related APIs. Our system newly discovers 25 APIs from Canvas, and 10 from WebGL(2). We find `WebGLRendering-Context.isEnabled` can be used to test whether a given WebGL capability is enabled. WebGL2 provides more APIs that tell its default settings than WebGL and it makes WebGL2 more attractive for fingerprinting abuse.

Performance. `Performance` APIs are commonly used in fingerprinting scripts. The most common way is to measure the execution time for certain operations. On different devices, since devices' hardware are different, their computational power can be distinguishable. In performance collected by our system, `Perfor-mance.measure` can be abusive in this way.

```
for every API in APIs:
    if isFP(API):
        API.weight = 1
    else:
        API.weight = 0
for every API in APIs:
    if API.weight == 1:
        counter += 1
    else:
        dummy.weight = counter
        delete previous APIs if their weights
        is 1
        counter = 0
for every dummy API:
    left_itr, right_itr = dummy
    left_weight, right_weight = dummy.weight-1
    while left_weight != 0:
        left_itr = left(left_itr)
        left_itr.weight += left_weight
        left_weight -= 1
    while right_weight != 0:
        right_itr = right(right_itr)
        right_itr.weight += right_weight
        right_weight -= 1
Sort every non-dummy APIs from high to low
return top x APIs
```

Listing 1: Locality Pseudocode

```
"addEventListener"in window&&window.addEventListener
("pageshow",function(t){t.persisted&&i()},!1)
```

Listing 2: PageTransitionEvent.persisted

```
isSmartPhone: (s = window.matchMedia && window.match
Media(" only screen and (min-device-width : 320px) a
nd (max-device-width : 480px)").matches || /(iPhone|
iPod)/g.test(navigator.userAgent), function() {retur
n s})
```

Listing 3: MediaQueryList.matches

`Performance` APIs provide timings for connection, response, navigation, and content loading. Although FP-RADAR noticed this type of abuse, they only label a small set of features due to their system limitation. Our system addressed the limitation and newly discovered 17 APIs in total. Besides time-related `Performance` API cases, `Performance.memory` interface tells information about heap memory, namely maximum JS heap size limit, total heap size, and heap size currently being used. FingerprintingAlert observed data related to these three APIs but provided no explanation.

D CASE STUDY SCRIPTS

```
I=a.screen;I&&(w.push(U("u_h",I.height)),w.push(U("u
_w",I.width)),w.push(U("u_ah",I.availHeight)),w.push
(U("u_aw",I.availWidth)),w.push(U("u_cd",I.colorDept
h)));a.history&&w.push(U("u_his",a.history.length))}
Z&&"function"==typeof Z.getTimezoneOffset&&w.push(U(
"u_tz",-Z.getTimezoneOffset()));b&&("function"==type
of b.javaEnabled&& w.push(U("u_java",b.javaEnabled()
```

Listing 4: History.length

```
{for(var e=["innerHeight","innerWidth","outerWidth",
"outerHeight","devicePixelRatio"],t={},r=0;r<e.lengt
h;r++){var n=e[r];t[n]=window[n]}return window.statu
sbar&&(t.statusbar_visible=window.statusbar.visible)
,t.length=window.length,t.modified=Object.getOwnProp
ertyNames(window.screen),JSON.stringify(t)}
```

Listing 5: Window.statusbar and BarProp.visible

```
a.deviceMemory:0,hardwareConcurrency:a.hardwareConcu
rrency?a.hardwareConcurrency:0,serviceWorkerStatus:"
serviceWorker"in a?a.serviceWorker.controller?"contr
olled":"supported":"unsupported"}:{}}......return{fe
tchTime:i-t.fetchStart,workerTime:t.workerStart>0?i-
t.workerStart:0,totalTime:i-t.requestStart,downloadT
ime:i-e,timeToFirstByte:e-t.requestStart,headerSize:
t.transferSize-t.encodedBodySize||0,dnsLookupTime:t.
domainLookupEnd-t.domainLookupStart}}
```

Listing 6: ServiceWorkerContainer.controller

```
if(t={},n=["accelerometer","accessibility-events","a
mbient-light-sensor","background-sync","camera","cli
pboard-read","clipboard-write","geolocation","gyrosc
ope","magnetometer","microphone"......],i=[],navigat
or.permissions)for(o in r=function(e){varr=n[e];i.pu
sh(navigator.permissions.query({name:r}).then(functi
on(e){t[r]=e.state}).
```

Listing 7: PermissionStatus.state and Permissions.query

E COMPARISON WITH FP-INSPECTOR

In 34(68%) features that are executed more than once during crawling, there are two major groups of features, WebGL(2) and EventHandler under the `Window` interface. In our seed list, there are only four WebGL(2) APIs and one API under the `Window` interface. They are not enough to cover all fingerprinting patterns in the wild. Thus, after adding fingerprinting APIs (under WebGL(2) and Window interfaces) from the APIs discovered by our system to the seed, our system can cover all 34 APIs.

```
navigator.mediaDevices.enumerateDevices().then(funct
ion(s){for(var v=0;v<s.length;++v){var p=s[v];"video
"===p.kind?p.kind="videoinput":"audio"===p.kind&&(p.
kind="audioinput"),p.deviceId?p.id||(p.id=p.deviceId
):p.deviceId=s.id,-1===c.indexOf(p)&&c.push(p)......
```

Listing 8: MediaDevices.enumerateDevices

F ROBUSTNESS OF THE SYSTEM

We did not observe any attack pattern during manual analysis. If we observe attackers inject random API calls, we can filter them out based on our dataflow analysis described in Section 4 and then run the locality algorithm. Injecting API calls with dataflow connections raises significantly the bar for attackers. Assuming attackers overcome it and inject one random API call (or any fixed number) between each fingerprinting API call, we still can filter them out from the sequence in the VV8 post-processor.

G BOT DETECTION WITH FINGERPRINTING

As fingerprinting is used to uniquely identify online users, crawlers (i.e. bots) are largely used to grab online information that can be identified by fingerprinting techniques. There are two main ways to identify crawlers, static features, and dynamic behavior. `Window.frames` and `Window.clientInformation` return abundant information which reveals specific sets of crawlers' settings. On the other hand, dynamic behavior often requires complicated analysis. Some naive crawlers interact (e.g. move the mouse) in a humanly impossible way or have no interaction. `IntersectionObserver` can be specifically used for behavior analysis. However, as indicated in Section 3.1, the accuracy of fingerprinting increases as more information is collected. Detecting crawlers based on a large list of fingerprinting APIs is more reliable than using a single API.

H ANALYSIS OF DISCOVERED FINGERPRINTING APIS

As shown in Section 6.2, discovered fingerprinting APIs can be used for fingerprinting script detection. Using these APIs is reliable since they are executed and the collected information is stored or transmitted by trackers. Each API returns different entropy. A reasonable approach is to combine fingerprinting APIs with their corresponding entropy. However, the crawling-based system is not able to collect users' fingerprints and calculate entropy. With enough fingerprints, the entropy of each fingerprinting API and an upper bound entropy that prevents users from being uniquely identified can be calculated. Based on these calculations, runtime fingerprinting detection and prevention are possible.

I AVAILABILITY

We make our source code, raw data, discovered APIs, fingerprinting code snippets, and the fingerprinting demonstration website publicly available.