FPFlow: Detect and Prevent Browser Fingerprinting with Dynamic Taint Analysis

Tianyi Li[1], Xiaofeng Zheng[2], Kaiwen Shen[2], and Xinhui Han[1(✉)]

[1] Peking University, Beijing, China
{litianyi,hanxinhui}@pku.edu.cn
[2] Tsinghua University, Beijing, China
{zxf19,skw17}@mails.tsinghua.edu.cn

Abstract. Browser fingerprinting is a practical user tracking technology widely adopted by many real-world websites to potentially track users' browsing behaviors. By collecting information such as screen resolution, user agent, and WebGL rendered data, the tracker can generate a unique identifier for users without their knowledge, leading to a severe violation of user privacy. Therefore, an effective detection and defense technology for browser fingerprinting is needed to protect user privacy. In this paper, we proposed FPFlow, a dynamic JavaScript taint analysis framework to detect and prevent browser fingerprinting. FPFlow monitors the whole process of browser fingerprinting, including collecting information, generating fingerprinting, and sending it to the remote server. We evaluated FPFlow on TRANCO top 10,000 websites. Our experiments showed that our framework could effectively detect browser fingerprints. We found 66.6% of the websites performing fingerprinting and revealed how browser fingerprinting is applied in real-world websites. We also showed that FPFlow could prevent browser fingerprinting with an acceptable overhead.

Keywords: Browser fingerprinting · Taint analysis · Privacy-enhancing technology

1 Introduction

Browser fingerprinting [21] is an online user tracking technique that collects a vector of browser-specific information, such as user agent, screen resolution, and installed browser fonts, etc., to uniquely identify the target browser. Previous studies [15,22] showed that the uniqueness of browser fingerprint could be as high as 89.4%. When combining hardware features by performing rendering tasks with HTML Canvas API and WebGL, browser fingerprint can even track users across browsers. Cao et al. [12] showed that they could uniquely identify more than 99% of 1,903 devices with 31 WebGL rendering tasks.

W. Lu et al. (Eds.): CNCERT 2021, CCIS 1506, pp. 51–67, 2022.
https://doi.org/10.1007/978-981-16-9229-1_4

Browser fingerprinting is widely used in several scenarios, such as personalized content and targeted advertising. The widespread deployment of tracking or user analyzing scripts allows trackers to track users across websites. Since browser fingerprinting is stateless (does not rely on client-side storage of identifiers), it is hard to detect and mitigate. Moreover, even the private mode of browsers cannot prevent browser fingerprinting.

Existing fingerprinting detection and prevention methods rely on pre-defined rules or known scripts [7,8,16,17]. Yet, not all of the prevention methods actually "protect" users [14], and some may even make the browser easier to be fingerprinted [10]. Prevention methods like Tor browser [6] will sacrifice user experience(e.g., disable HTML Canvas API and fix the window size). Modern browsers like Firefox have carried out countermeasures against browser fingerprinting. However, we found that Amiunique [1], a website investigating browser fingerprinting, can still uniquely identify the latest version of browsers. It is in dire need of an approach to detect and prevent browser fingerprinting, which motivated us to conduct this research.

Our Study. In this paper, we consider a website as performing browser fingerprinting if it collects fingerprinting attributes and sends them to the remote server. We proposed FPFlow, a dynamic taint analysis approach to detect and prevent browser fingerprinting, leading to potential violation of user privacy. FPFlow marks fingerprinting related attributes as taint source. During JavaScript execution, FPFlow propagates taint between the objects. When JavaScript tries to initiate a web request that carries taint, FPFlow considers it as a fingerprinting request and can block it.

We conducted a large-scale measurement study on TRANCO top 10,000 websites, a more reliable ranking list than Alexa [27]. Our result showed that 6,661 websites transmitted fingerprinting attributes. We further analyzed the fingerprinting attributes used in real-world websites and the behaviors of the tracking scripts.

Contributions: Our main contributions are:

- We proposed a data flow-based method to detect and prevent browser fingerprinting by monitoring all potential fingerprinting data transmission.
- We implemented FPFlow, an in-browser dynamic JavaScript taint analysis framework to detect and prevent browser fingerprinting with an acceptable overhead of 9.2%.
- We performed a large-scale analysis on TRANCO top 10,000 websites. We found 66.6% of websites are sending out fingerprinting attributes, and discussed the behaviors of the related scripts.

2 Related Work

Browser Fingerprinting. Browser fingerprinting is a method to identify a web browser without a stateful identifier like Cookie. Browser fingerprint is generated with a set of browser attributes such as user agent and screen resolution.

Eckersley [15] conducted the Panopticlick experiment in 2010. He used browser properties such as user agent, cookie-enabled to generate fingerprints, and used Flash or Java applets to probe system fonts. Among 286,777 fingerprints collected, 94.2% of them are unique when Java or Flash is enabled. Other browser properties such as battery status [26], installed fonts [18], and extensions [28–30] can also be used as browser fingerprinting.

Hardware features can also be used as part of browser fingerprinting. HTML Canvas and WebGL are widely studied as browser fingerprints representing hardware features. Mowery et al. [23] and Acar et al. [7] showed that rendered data in HTML Canvas has slightly difference in different machine or browser that can be used as browser fingerprinting. Cao et al. [12] carefully designed 31 WebGL rendering tasks and can identify 99.24% of users in their experiment. Englehardt et al. [16] discovered AudioContext based browser fingerprinting when crawling Alexa top sites for online tracking behavior analysis. They tested the feasibility of AudioContext based browser fingerprinting and found 713 different fingerprints among 18,500 users.

Detection of Browser Fingerprinting. To understand browser fingerprinting prevalence in the real-world, existing work proposed different methods to detect browser fingerprinting. Nikiforakis et al. [25] discovered 0.4% of websites in Alexa top 10,000 sites performing fingerprinting by looking for three known fingerprinting scripts.

Several works studied the adoption of browser fingerprinting by monitoring JavaScript APIs. Acar et al. [8] performed a large-scale study of browser fingerprinting on Alexa top 1 million sites. They modified the rendering engine to capture access to browser properties that can be used to perform browser fingerprinting. In 2014, Acar et al. [7] performed a large-scale study on Canvas fingerprinting. They monitored the calls and returns to Canvas API to decide whether a website performs browser fingerprinting and found 5,542 out of 100,000 sites were performing Canvas fingerprinting. Englehardt et al. [16] crawled Alexa top 1 million websites by monitoring the access to JavaScript native APIs. They found 14,317 sites performing Canvas fingerprinting and 67 sites performing AudioContext fingerprinting.

Al-Fannah et al. [9] crawled Majestic 10,000 sites and checked the web requests sent out by browser. A website is defined as engaging fingerprinting if at least one of 17 properties is present in the requests. They identified 6,876 sites that were performing browser fingerprinting.

Iqbal et al. [19] used a machine learning method to detect browser fingerprinting scripts. They extracted the AST of scripts and runtime API accesses as features and found that 22.7% of Alexa top 10,000 websites were performing browser fingerprinting.

Prevention of Browser Fingerprinting. To mitigate browser fingerprinting, Torres et al. [31] introduced FPBlock, a framework to generate a new fingerprinting for each visited domain to prevent cross-domain tracking. FPRandom [20], PriVaricator [24], and Disguised Chromium [11] are frameworks that prevent fingerprinting by randomizing browser properties or Canvas data.

FaizKhademi et al. [17] proposed FPGuard. Their framework first detected browser fingerprinting with 9 metrics. If suspicious behavior is detected, FPGuard will modify the content of the fingerprint. Modern browsers have also come up with fingerprinting protection strategies these years. Firefox blocks fingerprinting related scripts with a tracking script list [2] to protect user from browser fingerprinting.

Although various prevention methods have been proposed in academic research, not all of them can actually "prevent" browser fingerprinting. Vastel et al. [32] developed FP-Scanner to explore the inconsistencies of browser fingerprinting to detect potential alters to fingerprinting attributes. Datta et al. [14] evaluated 26 anti-fingerprinting tools and showed that not all of those protection methods are equal. Azad et al. [10] showed that all tools that attempt to modify the JavaScript behavior are unique fingerprintable, which makes the browser easier to be fingerprinted.

3 Motivation

Browser fingerprinting is a complex process in the JavaScript execution context. Different fingerprinting scripts collect different properties and call different functions to generate fingerprints. We call properties or functions used in browser fingerprinting **fingerprinting attributes**. The fingerprint is generated on the client-side and sent to tracking services through network requests. We call these requests **fingerprinting requests**. To better understand browser fingerprinting, we split the process of browser fingerprinting into five stages, as shown in Fig. 1.

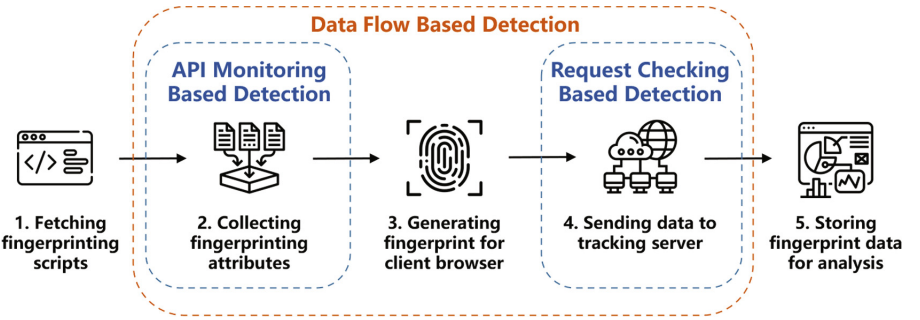

Fig. 1. The process of browser fingerprinting.

Previous studies working on Canvas-based fingerprinting detection and prevention [7,8,16,17] rely on rules defined by researchers. They monitor the access to specific APIs on stage 2, but they could not confirm that the rendered data is sent to the remote server. Their methods may lead to false-positive because Canvas and WebGL are more and more widely used in real-world websites. Besides, their methods cannot detect browser attributes based browser fingerprinting(e.g., the collection of user agent) because these attributes are likely to be accessed in benign scripts. Al-Fannah's study [9] checks whether the requests

sent to remote server contains fingerprinting attributes in stage 4. However, their work relies on the value of fingerprinting attributes. As a result, their work cannot detect fingerprint encoding, which will miss some websites that performing fingerprinting. Nor can they detect Canvas based fingerprinting because the canvas data is known before rendering.

In conclusion, existing work only focuses on a single stage of browser fingerprinting. API monitoring based approaches focus on stage 2, and requests checking-based approaches focus on stage 4. These methods do not take data flow into consideration, so the accuracy and ability of fingerprint detection are limited.

To fill the gap, we first define the browser fingerprinting behavior. **We consider a website performing browser fingerprinting if the client-side JavaScript code collects fingerprinting attributes and sends them to the remote server.**

Note that websites may collect fingerprinting attributes for benign reasons like user-agent statistics and language adaption. However, these websites are still capable of tracking users with the collected data. Besides, the information needed for providing client-side functionalities like user-agent and language can be obtained in HTTP headers, which does not depend on JavaScript execution. The website does not need to extract them from JavaScript context and send them to the remote server, especially the third-party ones. As a result, we consider websites that match our definition are all potentially involving browser fingerprinting.

Based on the definition, we introduced data flow analysis to help detect and prevent browser fingerprinting. We implemented a dynamic taint analysis framework FPFlow. FPFlow is a modified Chromium browser. It marks all fingerprinting attributes as taint source and all web request related functions as taint sink. FPFlow tracks the full life cycle of fingerprinting attributes from stage 2 to stage 4, and it can detect both browser attributes based fingerprinting and Canvas based fingerprinting. Our framework can recognize fingerprinting requests and intercept them before sending them out to prevent browser fingerprinting.

4 Technique Approach

In this section, we introduce the technique approach of FPFlow. We first give an overview of FPFlow in Sect. 4.1 to help understanding how FPFlow works. The following parts of this section explain the implementation of FPFlow in detail. Section 4.2 introduces the taint source and taint sink marked by FPFlow. Section 4.3 introduces the taint table and taint name table used to store object taints. Section 4.4 introduces bytecode instrument for runtime taint propagation in FPFlow.

4.1 Overview

Figure 2 shows the abstract architecture overview of FPFlow. FPFlow extends the JavaScript engine V8 and DOM engine Blink of Chromium with taint

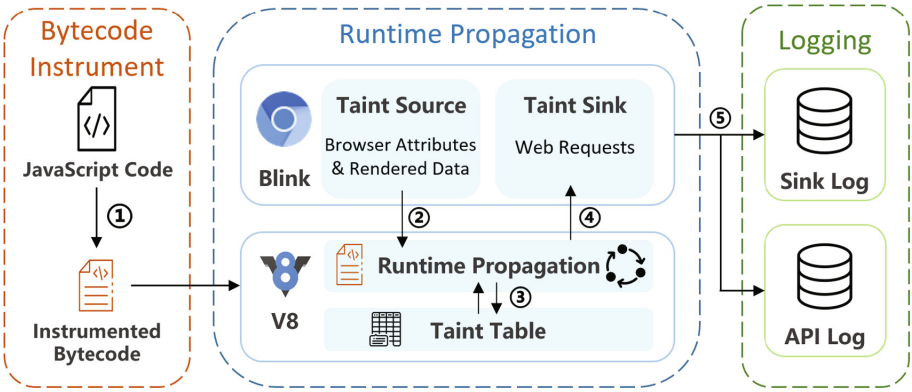

Fig. 2. Abstract architecture of FPFlow.

tracking capabilities. (1) When FPFlow visits a website, it first instruments the bytecode generated by the V8 engine to enable taint propagation. The instrumented bytecode is then executed by the V8 engine. (2) When JavaScript accesses the fingerprinting-related APIs in DOM, the V8 object is marked as tainted. (3) During the script execution, FPFlow propagates the taint between JavaScript objects and updates the taint table and taint name table. (4) When JavaScript tries to send a web request, FPFlow will check if the request carries taint before performing it. (5) If the URL or body of the request carries taint, a taint sink is triggered, and the corresponding log is generated. FPFlow will intercept such requests to prevent browser fingerprinting.

4.2 Taint Source and Taint Sink

The DOM interface of Blink is defined in WebIDL[1] files. WebIDL files define the properties and functions of DOM API. The V8-Blink binding code is generated according to WebIDL files from the code templates. FPFlow marks the taint source and sink properties or function in WebIDL files and modifies the code templates to hook the access to the taint source and sink.

FPFlow marks all fingerprinting attributes as taint sources. Fingerprinting attributes can be a property of the DOM element (e.g., `Cookie`) or the return value of a function (e.g. `toDataUrl`). When V8 tries to access the fingerprinting attributes, Blink will return an object to V8 that holds the value. If the attribute is marked as tainted, the return value is tainted with the name of the attribute.

Web tracking services need to collect user's fingerprints to identify users. Thus, the network request is a key step in browser fingerprinting that leads to privacy threats. FPFlow marks all network-related functions as taint sink. To prevent browser fingerprinting, FPFlow checks whether the request URL or body contains taint before the requests are actually processed. If the request contains

[1] https://heycam.github.io/webidl/.

taints, it is a fingerprinting request, and a taint sink is triggered. To prevent browser fingerprinting, FPFlow checks the taints carried by the request. If the request is recognized as an fingerprinting request, FPFlow skips the original request and returns an `undefined` object directly.

The example of marked taint source and taint sink is shown in Table 1. FPFlow marks 72 fingerprinting attributes as taint source (70 browser properties and 2 JavaScript functions) and 5 functions as taint sink. A full list of taint sources is available at https://github.com/FPFlow/FPFlow-project.

Table 1. Selected taint source and sink.

Type		DOM APIs
Source (72 in total)	Properties	`userAgent`, `innerHeight`, `colorDepth`, `Cookie` etc.
	Functions	`toDataUrl`, `getChannelData`
Sink (5 in total)		`XMLHTTPRequest`, `HTMLElement.src`, `WebSocket`, `Fetch`, `Navigator.sendBeacon`

4.3 Taint Table and Taint Name Table

To record the taint carried by JavaScript objects, FPFlow maintains a taint table, a hash table keyed on the internal addresses of V8 objects in each V8 instance. Once an object is tainted, FPFlow will add the object into the taint table along with its taint. When taint is propagated from an object to another object, the data in the object table is updated. As FPFlow uses object address as the key of taint table, there two special cases that need to be handled [13].

First, V8 garbage collection may move objects in memory. When the object is destructed or moved by V8 GC (garbage collection), FPFlow deletes or updates the corresponding entry in the taint table.

Second, Smi is a special type of JavaScript object in V8, which represents integers between -2^{30} and $2^{30} - 1$. The address allocation pattern for Smi is different from other types of objects in V8. Smi objects with the same value share one address (e.g., all Smi objects with value 0x14 shares address 0x1400000000). This feature optimizes the JavaScript runtime performance, but it causes over-taint in our system. FPFlow solves this problem with two steps. Firstly, FPFlow ensures that all values from the taint source are not Smi. If the taint source's value is a Smi value, FPFlow will convert it to HeapNumber, another number representation in V8. Secondly, FPFlow stops the conversion from any other type to Smi. Our method ensures that any object that carries taint cannot be Smi, and only introduces a slight performance overhead.

To accelerate taint propagation, the taint carried by an object is represented as a bitset. A bit in the bitset represents a certain kind of taint. If the object carries the taint, the bit in the bitset is set to 1. The taint propagation operation can be simplified to the logic or operation. We maintain a taint name table, which maps the string name of taint to the specific bit in the bitset. FPFlow maintains

one object taint table and one taint name table in each V8 instance to avoid conflict.

4.4 Taint Propagation

Once Chromium receives JavaScript source code, V8 parses the source file and generates the corresponding abstract syntax tree (AST). Then V8 generates bytecode according to the AST. We implement taint propagation logic by instrument additional bytecode in the V8 bytecode generation phase. The taint propagation logic is wrapped in V8 runtime functions and called through a single bytecode `CallRuntime`. Parameters related to taint propagation are passed to the runtime function through registers. FPFlow considers direct taint propagation in the following scenarios:

- Property load: If object `a` is tainted, the properties of `a` like `a.length` carries taint.
- Basic operations: Basic operations include mathematical operations, bit operations, logic expression. If one of the operands carries taint, the result of the operation carries taint as well.
- Native function call: The native function calls in JavaScript are implemented in C++. We need to propagate the corresponding taint when these functions are called. These function includes `encodeURIComponent`, `JSON.stringify`, `toString`, etc. If the parameter passed to the native function carries taint, these functions' return value also carries taints. We extract the address of the native functions during V8 bootstrap and check if a called function is a native function by comparing the function address.

An example of taint propagation is shown in Fig. 3. On line 1, the script gets the value of `navigator.vendor`, the object that holds the value of variable `x` carries the taint with name "navigator.vendor". On line 2, the taint is propagated from `x` to `y` because of the call to native functions. On line 3, the taint is propagated from `y` to `t` because of binary operation add. On line 4, the script tries to initiate a web request. The URL of the request (variable `t`) carries taint, so a taint sink is triggered. FPFlow will log the sink event and intercept this request.

```
1. var x = navigator.vendor;            // Visit Taint Source
2. var y = encodeURIComponent(x);       // Propagate for native function
3. var t = "https://tracker.com/?v="+y; // Propagate for binary operator ADD
4. (new Image).src = t;                 // Trigger taint sink
```

Fig. 3. Example of taint propagation.

4.5 Logging

FPFlow monitors all taint sinks and accesses to DOM API. For a taint sink, FPFlow records its request method, target URL, the taint carried by the request and the stack trace of the request. Each entry in the stack trace contains the function name, the JavaScript file it belongs to and the line number. For API access, FPFlow records the name of the accessed API and the access time.

5 Evaluation

In this section, we describe the experimental setup and present the result of applying FPFlow on TRANCO top 10,000 websites. We did not use Alexa list because previous research showed that Alexa rank is not stable and it changes daily up to 20%, which makes comparability of results difficult [27]. We were able to analyze the adoption of browser fingerprinting in those websites with FPFlow. We found 66.6% of the websites transmitting browser fingerprinting, which leads to potential browser fingerprinting based tracking. We also measured the effectiveness of FPFlow in preventing browser fingerprinting.

5.1 Experimental Setup

We crawled the homepage of TRANCO top 10,000 sites and gathered their behavior with FPFlow. FPFlow was driven by puppeteer [5] for automatically testing. To avoid the interference between websites caused by cookies or browsing history, we used a new browser instance for each website during the crawling process.

We captured all the script data during the crawling process. We used mitmproxy [4] to intercept all requests to JavaScript files and stored them for further analysis. In addition, we also used js-beautify [3] to format all JavaScript files captured by mitmproxy so that FPFlow could get a clear stack trace when the taint sink is triggered.

We waited for 120 s on each website during crawling to capture as many requests as possible. Meanwhile, we need to leave enough time for JavaScript code formatting since the size of loaded JavaScript code could be very large in modern web applications.

5.2 Large Scale Experiment Result

The crawling process took 30 h to complete. The detailed result is available at https://github.com/FPFlow/FPFlow-project. During the experiment, 40 websites (0.4%) did not work properly (e.g., did not respond or returned an HTTP error). In 9,960 successful crawled websites, 6,661 sites collected fingerprinting attributes from user browsers and sent them to remote servers. We refer to such websites as **fingerprinting websites**. Among the 6,661 fingerprinting websites, 6043 sent user data to the third-party domain, while 2,094 sent data to both first-party and third-party domains.

Table 2. Usage of tracking services.

Tracker domain	Sites
doubleclick.net	5,208
google-analytics.com	4,333
google.com	2,006
googlesyndication.com	1,370
rubiconproject.com	1,354
facebook.com	1,274
adnxs.com	754
rlcdn.com	596
casalemedia.com	581
criteo.com	493

Table 3. Usage of attributes.

Attribute	Used sites
UserAgent	6,397
Cookie	6,346
AppVersion	4,512
History:Length	4,473
Resolution	3,072
Platform	3,010
NavigatorLanguage	2,952
CookieEnabled	2,948
Screen:ColorDepth	2,319
Navigator:NavigatorPlugins	2,204

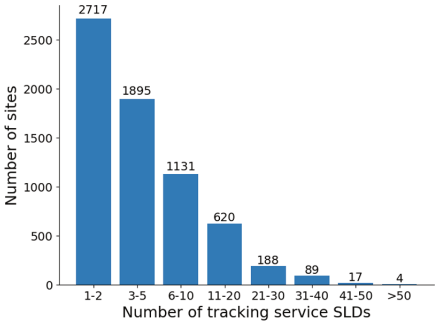

Fig. 4. Number of tracking services.

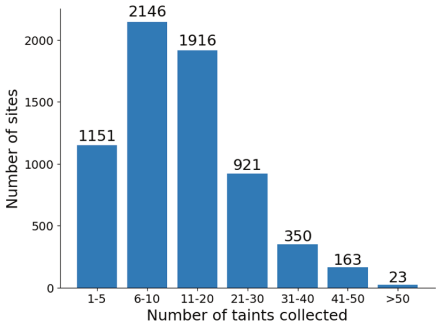

Fig. 5. Number of attributes collected.

We analyzed the tracking services that fingerprinting requests are sent to. Table 2 shows the most frequently used tracking services. We found that the most commonly used browser fingerprinting service providers are Google. We found 4,900 websites sending data to Google related domains[2].

We find that 5,363 websites send user data to more than one fingerprinting service. Figure 4 shows the number of tracking services used by TRANCO websites. Among 6,661 fingerprinting websites, the fingerprinting attributes were sent to an average of 5.59 domains. The maximum number of domains the fingerprinting attributes were sent to is 74.

Fingerprinting Attributes Collected by Tracking Services. The most used fingerprinting attributes are shown in Table 3. The widely used attributes are widely discussed in previous researches [15,25].

[2] google-analytics.com, doubleclick.net, google.com, googlesyndication.com.

We found that `document.cookie` is widely used in browser fingerprinting. This is because those tracking scripts are likely to store user fingerprint data in cookie data. The tracking service provider can track fingerprint changes in the client browser caused by software, operating system, and hardware update.

For rendering-based fingerprinting, we found 713 websites performing Canvas fingerprinting (including WebGL fingerprinting). Although previous research showed that Canvas fingerprinting could achieve high accuracy, it is not widely used in real-world websites, probably due to compatibility or performance reasons. We extract the initiator script of requests carry Canvas data. By manually inspecting the scripts, we found that Canvas fingerprinting scripts deployed to real-world websites mainly based on two open-source projects *fingerprintjs*[3] and *Picasso based canvas fingerprinting*[4]. By matching the keyword `fingerprintjs` and `picasso-like-canvas`, we found 334 websites use *the fingerprintjs* library, and 192 websites use *the Picasso* library.

We also find an obfuscated canvas fingerprinting script[5] is used by 88 websites. Our experiment found 49 websites performing AudioContext based fingerprinting.

We use the API trace recorded by FPFlow and filter rules from previous research to find Canvas-based font probing. More specifically, a website is performing Canvas-based font probing if it sets the `font` property on a Canvas to more than 20 different fonts and calls `measureText` over 20 times. We found 331 sites use Canvas-based font probing. We also found scripts using CCS-based font probing. This method is a part of *fingerprintjs* library. It first creates a `span` element, fills in some text, and sets a default font for text in span. To check if a font `F` is supported, the script creates another `span` element, fills in the same text, and sets the font to `F`. If the width and height of the two `span` elements are different, font `F` is supported in the browser.

The number of fingerprinting attributes collected by each website is shown in Fig. 5. A website collects 13.74 attributes on average, and the maximum number of collected taints is 55.

Request Methods. Table 4 shows the fingerprinting request methods used by TRANCO websites. We found that the most used request method is GET request with `src` attribute and POST request with `XMLHTTPRequest`. We also found that `Fetch`, `SendBeacon`, and `WebSocket` are also widely used in fingerprinting scripts, which is not mentioned in previous browser fingerprinting research.

Fingerprinting Initiator Scripts. We extract the initiator scripts of fingerprinting requests by analyzing the stack trace when the taint sink is triggered and comparing the initiator script with the target of the requests. Our result shows that the most used scripts are from the top user tracking services. We analyzed the number of different domains that a tracking script sends requests to. Most of these scripts initiate requests to a single domain. Some scripts will

[3] https://github.com/fingerprintjs/fingerprintjs.
[4] https://github.com/antoinevastel/picasso-like-canvas-fingerprinting.
[5] https://www.zalando.de/akam/11/2a40e12f.

Table 4. Fingerprinting request methods.

Request method	Used sites
XMLHTTPRequest	5,999
Element.src	4,795
Fetch	1,600
sendBeacon	1,319
WebSocket	128

send user data to several related domains. For example, scripts from google will send data to `google-analytics.com`, `google.com`, `googlesyndication.com` and `doubleclick.net`. We also find that some scripts try to load many different tracking scripts. We refer to these scripts as **tracker loader**. For example, we found 78 websites use tracking service from `cdn.krxd.net`. Each website using this tracking service has a configuration script indicating what third-party tracking service needs to be loaded.

Figure 6 is an example of the configuration script from `cdn.krxd.net`. The loader script from `cdn.krxd.net` will load all third-party trackers into the web page. Besides, the third-party trackers loaded by `cdn.krxd.net` use an id generated by the loader script, and the id is shared with those third-party services. This means the third-party services can also track the user with the help of the tracking service provided by `cdn.krxd.net`.

```
[{
    "id": 6,
    "name": "Google User Match",
    "content": "var kuid = Krux('get', 'user'); ... new Image().src =  \
     'https://cm.g.doubleclick.net/pixel?google_nid=krux_digital&google_hm='+baseEncodedKuid"
}, {
    "id": 21,
    "name": "Acxiom",
    "content": "var kuid = Krux('get', 'user'); ... \
     var liveramp_url = 'https://idsync.rlcdn.com/379708.gif?partner_uid=' + kuid;"
}, {
    "id": 153,
    "name": "Datonics User Match",
    "content": "var kuid = Krux('get', 'user'); ... \
    var datonics_url = 'https://fei.pro-market.net/engine?mimetype=img&du=88&csync=' + kuid;"
},]
```

Fig. 6. A example of fingerprinting configuration script with shared user ID.

Fingerprinting Beacon. Our experiment shows that many sites sent user data to a URL many times. These requests contain the same or different parameters. We call a website sending **fingerprinting beacon** if it sends more than 5 requests to a single URL. We found 860 websites are sending fingerprint beacons, and 674 sites are sending out fingerprint beacons with different parameters each time.

We analyzed the stack trace of the beacon requests and extracted the function names in the stack. We searched for keywords `event` and `interval` in function names and found matched function names in fingerprinting beacon from 57 sites. The matched function names includes `postEvent`, `trackAnalyticsEvent`, `GoogleAnalyticsEventTracking` and `setInterval`. By manually analyze the web page, we found that these requests are triggered at regular intervals or when a specific event is triggered. For example, we found the script from `jd.com` add fingerprinting events to the logo of the page. The fingerprinting request is triggered when the user moves the mouse over the logo. This indicates that the tracking service is tracking the user's visit history and recording the user's detailed browsing behavior.

5.3 Evaluate the Accuracy of Taint Analysis

The lack of standard reference for browser fingerprint usage and the huge volume of front-end code make it difficult to analyze them all. Therefore, we evaluated the FPFlow detection results mainly by random sampling and manual verification.

We randomly selected 50 websites that use browser fingerprinting. These 50 sites contain 400 requests containing tainted requests. We analyzed these browser fingerprint requests manually. We determined whether the requests were false positives by analyzing the information carried in the requests and the logic of the script code that initiated the requests. In our manual analysis, we found 39 requests to be FPFlow false positives. The estimated false positive rate of FPFlow is 9.75%.

5.4 Fingerprinting Prevention

Collection of only a few fingerprinting attributes is not enough to generate a precise fingerprint for client user. As prevention method, we consider a request as fingerprinting request if (1) it is using Canvas based fingerprinting or AudioContext based fingerprinting, or (2) it carries more than 10 taints.

To test the usability, we tested the extended FPFlow by manually browsing the top 50 sites in the TRANCO list. We stopped for 1 min to perform basic operations for each site, like click the link and log in. The fingerprinting requests were successfully blocked, and we did not observe any abnormalities during the test.

Previous research [19] discussed the page breakage caused by request blocking. They stated that URL blocking-based protection could affect the user experience because request blocking will block the content loading. FPFlow won't block the resource loading request (e.g., loading content from a tracker or advertising domain). Instead, FPFlow only intercepts data transmission requests. We found that these requests seldom return data. For example, many fingerprinting requests using `src` attributes requests for a zero-size GIF image. Such requests are only used for collecting client data, and they do not load anything. As a result, blocking such requests will not cause breakage to the web page.

We also evaluated the overhead of FPFlow by comparing it with the original Chromium browser. Since the API access monitoring feature of FPFlow is only used to compare the result with previous work and introduces relatively large IO overhead, we disabled the API access monitoring feature in the performance testing. We selected the TRANCO top 100 sites and loaded them with FPFlow and original Chromium. We recorded the time from the start of the two browsers to the end of the page loading. The performance overhead ranges from 6% to 13%, with an average of 9.2%.

6 Discussion

Our Improvements to Previous Approaches. Comparing with API monitoring based detection and prevention [7,8,16,17], FPFlow can reduce false positives. We found radio garden[6], an online FM website using Canvas and WebGL to generate the background of the page. The API access trace of this website contains many operations related to WebGL, and the generated image data is retrieved through API `toDataURL`. It is likely for previous work to mistake this website as performing Canvas fingerprinting. However, FPFlow showed that the rendered data is not sent to the remote server.

```
getLocation: function() {
    return fingerprint.util.MD5.hex_md5(location.href.split("?")[0])
},
getUserAgent: function() {
    return fingerprint.util.MD5.hex_md5(navigator.userAgent)
}
.....
t.push("canvas fp:" + fingerprint.util.MD5.hex_md5(r.toDataURL())),
```

Fig. 7. Fingerprinting encoding script.

We found that encoding or hashing browser fingerprints is a common practice. The encoded fingerprinting is transmitted through the web or stored in Cookie as a user identifier. Comparing with request checking based detection [9], FPFlow is not limited by fingerprint encoding since encoding or hashing does not cut off taint propagation. Figure 7 is a formatted code snippet from https://wl.jd.com/wl.js. The fingerprinting data (including location, user agent, and rendered Canvas data) is hashed with the MD5 algorithm.

Limitations. Although FPFlow can detect fingerprinting attributes transmission, it has several limitations. First, FPFlow propagates taint only with explicit data flow, and it is not able to propagate with implicit data flow, which results in false negative. As a result, our experiment revealed the lower bound of the

[6] http://radio.garden/.

current deployment of potential browser fingerprinting. Second, FPFlow cannot detect WebRTC fingerprinting and JavaScript font probing because these techniques do not rely on the return value of certain API. These fingerprinting methods can be detected with the API accessing pattern, as mentioned in previous researches.

7 Conclusion

In this paper, we introduced FPFlow, a pure dynamic taint analysis framework upon Chromium to detect and prevent browser fingerprinting. FPFlow monitors the data flow from retrieving fingerprinting attributes to sending them to tracking service. Based on FPFlow, we conducted a large-scale browser fingerprinting detection on TRANCO top 10,000 sites and found that 66.6% of the websites are transmitting fingerprinting data, leading to potential fingerprinting based tracking. Meanwhile, our experiments revealed the behavior of fingerprinting scripts such as tracker loader and fingerprinting beacon. We also showed that FPFlow could prevent browser fingerprinting with no sacrifice to user experience. Our work introduces data flow analysis to have a better understanding of how browser fingerprinting is adopted in the real world.

References

1. Amiunique. https://amiunique.org/fp
2. Firefox's protection against fingerprinting. https://support.mozilla.org/en-US/kb/firefox-protection-against-fingerprinting
3. JS-beautify. https://github.com/beautify-web/js-beautify
4. mitmproxy. https://mitmproxy.org/
5. Puppeteer. https://pptr.dev/
6. Tor project. https://www.torproject.org/
7. Acar, G., Eubank, C., Englehardt, S., Juarez, M., Narayanan, A., Diaz, C.: The web never forgets: persistent tracking mechanisms in the wild. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 674–689 (2014)
8. Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., Preneel, B.: FPDetective: dusting the web for fingerprinters. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 1129–1140 (2013)
9. Al-Fannah, N.M., Li, W., Mitchell, C.J.: Beyond cookie monster amnesia: real world persistent online tracking. In: Chen, L., Manulis, M., Schneider, S. (eds.) ISC 2018. LNCS, vol. 11060, pp. 481–501. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99136-8_26
10. Amin Azad, B., Starov, O., Laperdrix, P., Nikiforakis, N.: Short paper - taming the shape shifter: detecting anti-fingerprinting browsers. In: Maurice, C., Bilge, L., Stringhini, G., Neves, N. (eds.) DIMVA 2020. LNCS, vol. 12223, pp. 160–170. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-52683-2_8

11. Baumann, P., Katzenbeisser, S., Stopczynski, M., Tews, E.: Disguised chromium browser: robust browser, flash and canvas fingerprinting protection. In: Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society, pp. 37–46 (2016)
12. Cao, Y., Li, S., Wijmans, E., et al.: (cross-) browser fingerprinting via OS and hardware level features. In: NDSS (2017)
13. Chen, Q., Kapravelos, A.: Mystique: uncovering information leakage from browser extensions. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1687–1700 (2018)
14. Datta, A., Lu, J., Tschantz, M.C.: Evaluating anti-fingerprinting privacy enhancing technologies. In: The World Wide Web Conference, pp. 351–362 (2019)
15. Eckersley, P.: How unique is your web browser? In: Atallah, M.J., Hopper, N.J. (eds.) PETS 2010. LNCS, vol. 6205, pp. 1–18. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14527-8_1
16. Englehardt, S., Narayanan, A.: Online tracking: a 1-million-site measurement and analysis. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1388–1401 (2016)
17. FaizKhademi, A., Zulkernine, M., Weldemariam, K.: FPGuard: detection and prevention of browser fingerprinting. In: Samarati, P. (ed.) DBSec 2015. LNCS, vol. 9149, pp. 293–308. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20810-7_21
18. Fifield, D., Egelman, S.: Fingerprinting web users through font metrics. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 107–124. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47854-7_7
19. Iqbal, U., Englehardt, S., Shafiq, Z.: Fingerprinting the fingerprinters: learning to detect browser fingerprinting behaviors. arXiv preprint arXiv:2008.04480 (2020)
20. Laperdrix, P., Baudry, B., Mishra, V.: FPRandom: randomizing core browser objects to break advanced device fingerprinting techniques. In: Bodden, E., Payer, M., Athanasopoulos, E. (eds.) ESSoS 2017. LNCS, vol. 10379, pp. 97–114. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62105-0_7
21. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser fingerprinting: a survey. ACM Trans. Web (TWEB) **14**(2), 1–33 (2020)
22. Laperdrix, P., Rudametkin, W., Baudry, B.: Beauty and the beast: diverting modern web browsers to build unique browser fingerprints. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 878–894. IEEE (2016)
23. Mowery, K., Shacham, H.: Pixel perfect: fingerprinting canvas in HTML5. In: Proceedings of W2SP, pp. 1–12 (2012)
24. Nikiforakis, N., Joosen, W., Livshits, B.: PriVaricator: deceiving fingerprinters with little white lies. In: Proceedings of the 24th International Conference on World Wide Web, pp. 820–830 (2015)
25. Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: Cookieless monster: exploring the ecosystem of web-based device fingerprinting. In: 2013 IEEE Symposium on Security and Privacy, pp. 541–555. IEEE (2013)
26. Olejnik, Ł, Acar, G., Castelluccia, C., Diaz, C.: The leaking battery. In: Garcia-Alfaro, J., Navarro-Arribas, G., Aldini, A., Martinelli, F., Suri, N. (eds.) DPM/QASA -2015. LNCS, vol. 9481, pp. 254–263. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29883-2_18
27. Pochat, V.L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., Joosen, W.: Tranco: a research-oriented top sites ranking hardened against manipulation. arXiv preprint arXiv:1806.01156 (2018)

28. Sjösten, A., Van Acker, S., Sabelfeld, A.: Discovering browser extensions via web accessible resources. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 329–336 (2017)
29. Starov, O., Laperdrix, P., Kapravelos, A., Nikiforakis, N.: Unnecessarily identifiable: quantifying the fingerprintability of browser extensions due to bloat. In: The World Wide Web Conference, pp. 3244–3250 (2019)
30. Starov, O., Nikiforakis, N.: XHOUND: quantifying the fingerprintability of browser extensions. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 941–956. IEEE (2017)
31. Torres, C.F., Jonker, H., Mauw, S.: *FP-Block*: usable web privacy by controlling browser fingerprinting. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015. LNCS, vol. 9327, pp. 3–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24177-7_1
32. Vastel, A., Laperdrix, P., Rudametkin, W., Rouvoy, R.: FP-scanner: the privacy implications of browser fingerprint inconsistencies. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 135–150 (2018)