Karlsruhe Institute of Technology

KASTEL

The Feasibility of Limiting Access to Fingerprintable Surfaces in Web Browsers

Bachelorarbeit

KIT – Karlsruher Institut für Technologie
ITI – Institut für Theoretische Informatik
Forschungsruppe Kryptographie und Sicherheit

Jakob Gretenkort

March 19, 2021

Verantwortlicher Betreuer: Prof. Dr. Müller-Quade
Betreuender Mitarbeiter: F. Dörre

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 19. März 2021

(Jakob Gretenkort)

Abstract

In this work, we examine the feasibility of limiting access to fingerprintable surfaces for the purpose of making browser fingerprinting ineffective for tracking users. Specifically, we aim to determine whether the Privacy Budget, as proposed by google as part of their Privacy Sandbox project, is fit for this purpose. The Privacy Budget monitors how much fingerprinting information is extracted from a browser installation by a website and stops further extraction of information once a threshold is reached.

We come to the conclusion, that the privacy budget, while feasible and fit for purpose, is faced with many problems that remain yet unsolved. As it is currently proposed, the Budget does not make use of the fact that fingerprints may be unstable. We try to push in this direction by introducing a new metric for fingerprint stability based on entropy. The budget also fails to take into account the possibilities of fingerprinting across multiple websites, which we detail in this work.

We survey a list of 10000 popular websites to examine which fingerprinting surfaces they use in order to establish whether the privacy budget would break them, leading to a failure in user acceptance. For this, we develop an application called FPLog, a modified browser which logs website's accesses to fingerprinting surfaces as they occur. Evaluating our results from visiting the 10000 websites with this application, we come to the conclusion, that current websites make use of many fingerprinting surfaces, but not so many as to make an entropy-based limit infeasible.

Zusammenfassung

In dieser Arbeit untersuchen wir die Möglichkeit, Browser Fingerprinting zum Verfolgen von Nutzern ineffektiv zu machen, indem der Zugang von Webseiten zu fingerprintbaren Browseroberflächen beschränkt wird. Insbesondere möchten wir bestimmen, ob das von Google als teil der von ihnen entwickelten Privacy Sandbox vorgeschlagene Privacy Budget sich hierzu eignet. Dieses Privacy Budget beobachtet wie viel Information zum Fingerprinten eine Webseite aus einer Browserinstallation gewinnt und schneidet der Webseite den Zugang zu weiteren Informationen ab, wenn sie einen gewissen Grenzwert erreicht.

Wir kommen zu dem Schluss, dass das Privacy Budget zwar umsetzbar und zweckmäßig ist, aber einige wichtige Fragen dazu aktuell noch geklärt werden müssen. Der aktuelle Vorschlag zum Budget sieht keine Nutzung der Tatsache vor, dass Fingerprints unter Umständen instabil sind. Wir versuchen, diese Richtung der Forschung voran zu treiben, indem wir eine neue Metrik für die Stabilität von Fingerprints einführen, die auf Entropie basiert. Das Budget beachtet außerdem aktuell nicht die Möglichkeit, Fingerprinting über mehrere Webseiten zu verteilen. Wir beleuchten diese Möglichkeit näher.

Wir untersuchen eine Gruppe von 10000 pupolären webseiten darauf, welche fingerprintbaren Browseroberflächen sie nutzen um zu bestimmen, ob die Einführung des Budgets sie in ihrer Funktion einschränken würde. Hierfür entwickeln wir FPLog, einen modifizierten Browser der Zugriffe auf seine Oberflächen live protokolliert. Die Ergebnisse des Besuchs der 10000 Websieten mit diesem Browser führen uns zu dem Ergebnis, dass aktuelle Webseiten zwar auf viele Fingerprintbare Oberflächen zugreifen, aber nicht so viele, dass entropiebasierte Einschränkungen wie die vom Privacy Budget nicht sinnvoll umzusetzen wären.

Contents

1 **Introduction** 1

1.1 Structure of this Work . . . 2

1.2 Related Work . . . 2

2 **Browser Fingerprinting** 3

2.1 Entropy and Recognizability . . . 4

2.1.1 Distinctiveness . . . 4

2.1.2 Stability . . . 5

2.1.3 Examples . . . 6

2.2 Fingerprintable Surfaces . . . 7

2.3 HTTP Headers . . . 7

2.4 Javascript . . . 8

2.5 Membership Query Fingerprints . . . 11

2.6 Media APIs . . . 11

2.6.1 Canvas . . . 12

2.6.2 Audio . . . 13

2.6.3 Video . . . 13

2.7 Fonts . . . 13

3 **Google's Privacy Sandbox** 15

3.1 Estimating Information . . . 16

3.2 Privacy Budget . . . 16

3.3 Involuntary access . . . 18

4 **Detecting Fingerprinting** 19

4.1 Detecting Access . . . 19

4.1.1 Media APIs . . . 20

4.1.2 Fonts . . . 20

4.2 Distinguishing fingerprinting and non-fingerprinting access . . . 21

4.3 Traceability of gathered Information . . . 21

4.4 Fingerprinting with multiple sites 22

5 Implementation 23

5.1 Javascript 23

5.2 Fonts 27

5.3 Hosts 27

5.4 Limitations of FPLog 27

5.5 Selecting Websites to Log 28

5.6 Automation 29

6 Evaluation 31

6.1 Fonts 31

6.2 Javascript 33

7 Lowering Surface Use 37

7.1 Compression 37

7.2 Compatibility 37

8 Conclusion 39

Bibliography 41

1 Introduction

When the internet was first conceived, it was built on a protocol known as HTTP, which is fundamentally stateless. This means that if a user made two requests to the same host in succession, the second request did not carry any information that would deliberately make the user recognizable to the host as the originator of the first request. This of course had the practical advantage that neither the user nor the host had to keep track of any session state. However, this also limited the types of applications hosts could create on the internet to simple static information retrieval.

This changed when the Netscape web browser introduced the notion of cookies. These allow hosts to send small text files to visitors which the visitor's browser will send along with every subsequent request to the same host as identified by the host's domain name.[1] If the content of such a file is a unique ID, the host can attach a meaningful session state to that ID on his end, effectively making website visits stateful.

Cookies also allow for user's activities to be tracked across visits to multiple sites if all of those sites contain resources retrieved from one common host. If during the visit to a website a cookie is sent to any host that is not the host of the site itself, we refer to this as a *third-party cookie*.

Today, the ability of third-party cookies to track users across multiple sites is often seen as problematic for reasons of privacy, because it allows hosts which have their content embedded on many popular websites, such as advertisers or content delivery networks, to build detailed profiles about users' interests and habits. To combat this use of cookies, many popular browsers have introduced options for users to limit or entirely forbid the use of third-party cookies in recent years.

All parties interested in tracking users across sites have since had to adopt new ways of achieving this, one of which is browser fingerprinting. With this technology, identifying information about a specific browser installation is gathered and compiled into a fingerprint. This occurs at a *browser's surfaces*, which are all points of interaction, at any level of decomposition,

1 It should be noted that cookies set for a domain are also sent along with requests to hosts for subdomains of that domain and even override any cookie of the same name that is set on that subdomain. Furthermore, subdomains are generally permitted to set cookies for their parent domains too. A notable exception are "public suffixes" such as "com" or "uk.co", for which many browsers refuse cookies for security reasons.[Bar11]

between a browser installation and a visited website and its hosts. If the information is stable across visits to multiple sites and unique among the set of all browser installations that visit those sites, the corresponding fingerprint can be used as a means of tracking the installation.

Fingerprinting is problematic for the same reasons as third-party cookies are. Therefore, browser vendors have started to develop mitigation strategies against fingerprinting, one of which is to limit the access that sites have to a browser's surfaces to a level at which a host can not gather enough information about a browser installation to generate a unique fingerprint. Many of these surfaces are however exposed to the host with good reason, and limiting access to them can limit the functionality of websites. This calls into question the feasibility of this strategy, which we will discuss in this thesis.

1.1 Structure of this Work

We begin by introducing the theory behind browser fingerprinting and how it works in chapter 2. The mitigation strategy against fingerprinting examined in this thesis is defined in chapter 3. Our approach to evaluating the feasibility of this strategy is explained in theory in chapter 4, our implementation of this approach is described in chapter 5 and our results are evaluated in chapter 6. Some websites that caught our attention during the evaluation are further examined in chapter 7, before we draw our conclusions in chapter 8.

1.2 Related Work

The perhaps most important surveys on the feasibility of fingerprinting in general are [Eck10] and [GLB18], which we cite throughout this work. Though we focus on limiting access to fingerprintable surfaces, which is a relatively new idea, there are of course other ways to combat fingerprinting which have been studied extensively. One of the easier ways to combat fingerprinting, which unfortunately also negatively impacts the browser's usability, is to randomize surfaces, which is studied in [LBM17].

2 Browser Fingerprinting

Any web browser installed on any device may possess "subtle but measurable variations"[Eck10] when compared to any other browser installation. These differences may arise from the browser's implementation or the libraries and hardware it relies upon. Some of these differentiating properties may be measured and used to generate an identifier or 'fingerprint' of the installation which may be unique among the set of all globally installed browsers [Eck10]. Even if it is not globally unique, the fingerprint may still be unique within a certain context, for example the set of all browsers to visit a site within the previous month [Eck10].

Such a unique fingerprint, if it is unchanging between visits to the website, can be used to identify the browser installation and track its user's activities without the need for the browser to provide any explicitly identifying information, such as cookies, to the site [Eck10]. If the fingerprint is generated exclusively with information from the browser that is independent of the visited website, the fingerprint may even be unchanging across visits to different domains, making it suitable for cross-site tracking [Eck10].

We will define any set of websites and hosts which gather fingerprints with the same fingerprinting method and exchange collected fingerprints among one another as a *fingerprinting network*. The usual implementation of such a network is as a single host having their fingerprinting method embedded on multiple sites, usually as a small javascript combined with some server-side measures. Note that websites and hosts can belong to multiple fingerprinting networks, though in practice this is much more likely for websites than hosts.

In order to be suitable for the purpose of reliably recognizing browser installations across interactions with a fingerprinting network, a method of fingerprinting must have the following properties:

- **Distinctiveness:** The method needs to gather enough identifying information to have a high chance of making a suitably large subset of visiting browser installations uniquely identifiable by their fingerprints.

- **Stability:** Two fingerprints taken from the same browser in succession must have a high chance of not having so many changes between them as to have become unrecognizable.

2.1 Entropy and Recognizability

Currently, distinctiveness and stability are defined rather vaguely, so we will try to introduce some metrics for both properties in order to then make their definitions firmer.

Let B be the global set of all browser installations.

A property of a browser installation $b \in B$ is *fingerprintable*, if and only if it can at all be measured as a value different from 'immeasurable' by at least one fingerprinting network visited by the browser.

Let S_b be the set of all fingerprintable properties of $b \in B$, and let $S := \bigcup_{b \in B} S_b$ be the set of all properties that are fingerprintable in at least one browser installation.

2.1.1 Distinctiveness

For any fingerprintable property $s \in S$, let $F_s(b), b \in B$ be a method of fingerprinting s on browser installations. Let the outputs of this fingerprinting method be distributed according to the discrete probability density function $P(f_{sn}), n \in [1, \ldots, N]$. Then, the information obtained from a fingerprint of a property is given by [Eck10]:

$$I(F_s(b) = f_{sn}) = -\log_2(P(f_{sn})) \tag{2.1}$$

Because of our choice of a base two logarithm, I is measured in bits. We can then calculate the average information across all f_{sn}, giving us the entropy H_s of a property's fingerprint [Eck10]:

$$H_s = H(F_s) = \sum_{n=1}^{N} P(f_{sn}) I(F_s(b) = f_{sn}) = -\sum_{n=1}^{N} P(f_{sn}) \log_2(P(f_{sn})) \tag{2.2}$$

From multiple property fingerprints F_s, we can construct a fingerprint F for the browser installation simply by concatenation. If the property fingerprints are statistically independent of one another, which is rarely the case [Eck10], the combined fingerprint's entropy can be calculated by simple addition:

$$H(F) = \sum_{s \in S} H_s \tag{2.3}$$

However, if the property fingerprints $F_{s1}, .., F_{sk}$ are not statistically independent, joint entropy has to be calculated like this:

$$H(F) = -\sum_{f_1 \in F_{s1}} \cdots \sum_{f_K \in F_{sK}} P(f_1, \ldots, f_K) \log_2(P(f_1, \ldots, f_K)) \tag{2.4}$$

Where $P(f_1, \ldots, f_k)$ is the joint probability distribution of the property fingerprints. It is important to note, that the combined fingerprint F will always contain at least as much information as its most informative contributing property fingerprint:

$$\forall s \in S \colon H(F) \geq H_s \tag{2.5}$$

This also means, that when adding a new property fingerprint to an existing combined fingerprint, the entropy of the combined fingerprint cannot decrease.

H is a good measure for a fingerprint's distinctiveness and can be interpreted intuitively: If fingerprints were perfectly stable, any one browser installation would on average share its fingerprint with only one in 2^H browser installations [Eck10].

2.1.2 Stability

An important metric for measuring the stability of a fingerprinting method is the chance of a fingerprint to change between two successive visits to a fingerprinting network. This metric is quite important for practical reasons, as any algorithm trying to track browser installations despite partial changes to their fingerprints will deal with exactly these incremental changes from one visit to the next.

In literature, for example [Eck10] and [Vas+18], a modified version of this metric is used, where instead of the chance of change between visits, the chance of change between points in time is examined. This metric is a little less useful for practical application, but has the advantage of being independent from the rate at which browsers visit a fingerprinting network.

These two metrics both have the problem of being quite hard to relate to distinctiveness, so we will define a new metric which measures the expected entropy of a property fingerprint within a browser installation based on its chance to change between visits.

For any fingerprintable browser property $s \in S$, let $V_s = \{v_{s_1}, \ldots, v_{s_n}\}$ be the set of all values across all browsers this property can have when fingerprinted by the arbitrary but fixed fingerprinting network FP.

We will assume V_s to be finite due to the limited number of browser installations that exists during the lifetime of FP.

When a browser installation visits a fingerprinting network, there is a chance for every fingerprintable property $s \in S$ of the installation to either have changed to a new value or to have remained the same. We will express this chance with the discrete probability density function $P_s(v,w) \colon V_s^2 \to \mathbb{R}$.

We will consider only those properties of browser installations that can always change from any value to any other:

$$\forall v, w \in V_s^2 : P_s(v,w) > 0 \tag{2.6}$$

As we will see later, this is usually the case for any 'useful' fingerprintable property, because they tend to be in control of the browser installation, which can of course always have its configuration changed arbitrarily. For example, even when fingerprinting a browser's version number, which usually always stays the same or increases, it is theoretically possible for a user to downgrade to any previous version.

If we interpret V_s as the states of a markov chain Y_s who's outputs are always equal to the state reached after a state transition, and have the state transition probabilities be defined by P_s, then this chain's outputs model the values of s we would expect to see when fingerprinting browser installations.

The entropy of the output of this markov chain, and therefore the entropy of changes to s within one browser installation can be calculated with

$$G_s = G(Y_s) = -\sum_{i,j=0}^{|V_s|} \pi_i P_s(v_i,v_j) \log_2(P_s(v_i,v_j)) \tag{2.7}$$

where π is the stationary distribution of Y_s. Because V_s is finite and (2.6), Y_s is trivially irreducible, aperiodic and recurrent. Therefore, π exists, is unique and is defined by:

$$\pi = \pi * M \tag{2.8}$$

where $M = (m_{ij}) = (P_s(v_{si}, v_{sj}))$ is the state transition matrix, and $\sum_{i=0}^{|V_s|} \pi_i = 1$.

G_s measures changeability, the inverse of stability.

2.1.3 Examples

Take for example a random number generator. If we tried fingerprinting a browser installation by requesting one random bit from it, this property fingerprint would have a distinctiveness of $H = 1b$ but also a stability of $G = 1b$. This points to the fact that if $G \geq H$, the fingerprinting method is too unstable to be useful. If we were to try fingerprinting by checking for the existence of javascript's `document` object, a distinctiveness of $H = 0b$ immediately tells us, that we are not getting useful information for fingerprinting. A more practical example is the browser vendor, accessible with javascript in `navigator.vendor`. If 20% of the fingerprinting network's visitors use Firefox (vendor = Mozilla), and 80% use Chrome (vendor = Google), the distinctiveness of this property will be $H = 0.722b$, which is not much, but the property fingerprint is proven to be worthwhile by a changeability of $G \approx 0$, because changes in browser

are quite rare.

2.2 Fingerprintable Surfaces

Recall that a browser surface is any point of interaction, at any level of decomposition, between a browser installation and a visited website and its hosts. A browser surface is *fingerprintable*, if it allows a fingerprinting network the measurement of a fingerprintable property that has a distinctiveness that is strictly greater than its changeability: $H > G$.

In general, there are two locations at which fingerprinting networks will extract information from fingerprintable surfaces. Host side surfaces typically include any part of the communication stack, especially IP, TCP and HTTP/S headers, while client side surfaces are mostly focused on javascript APIs. There used to be heavy use of Flash and Java in client side fingerprinting, but these methods are becoming increasingly irrelevant after Flash has been discontinued and most browsers disable Java by default [Ado21][Ora21].

2.3 HTTP Headers

When a browser requests web content from a host, it will usually do so via HTTP or HTTPS. In order to ensure that the received content will fit both the implementation of the browser and the preferences of its user, HTTP allows the browser to attach compatibility information and preferences to its requests in the form of HTTP headers. Much of the information in these headers is independent of the requested website and stable and distinctive enough for fingerprinting.

The most important examples for headers with good properties for fingerprinting are the accept headers (Accept ($H = 0.729$), Accept-Encoding ($H = 0.382$), Accept-Language ($H = 2.716$)), the Do-Not-Track header ($H = 1.919$) and the User-Agent header ($H = 7.150$) [GLB18].

The accept headers are fairly self explanatory, they give notice of content types and encodings the requesting browser installation can process, as well as the languages the browser's user can understand. The Do-Not-Track header signals a request from the user not to have their activities tracked, and is commonly ignored.

The User-Agent header is interesting for fingerprinting because it contains the user agent string, which is basically a name that browser installations give themselves in order to give hosts a rough idea of their capabilities and configuration. Take for example the following user agent string: "`Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36`". It identifies an installation

of Google Chrome in version `"88.0.4324.150"`, which is using a fork of the `AppleWebKit` engine in version `537.36` and is compiled for Windows 10 x64 .[1]

user agent strings do not have a clearly defined format and may contain less or more information than this. On Linux for example, the Chrome user agent string may contain an identifier of the used window system (e.g. `"x11"`). Also, some browsers such as Opera may allow users to choose a user agent string from a different installation for the browser to present itself with either for privacy reasons or to allow users to see content otherwise restricted to certain browser types.

The Accept-Charset header used to be a good source of entropy as well, but in a move against fingerprinting, many browser vendors discontinued support for it [Moz21]. It is important to understand why vendors were able to do this. The Accept-Charset header contains a list of text encodings understood by the browser installation that makes the request [FR14], and is meant to be used for browsers to communicate their capability to understand some uncommon encodings. Hosts may want to know of these capabilities, because they may be able to serve better content when they are able to make use of some encoding-specific special characters [FR14]. With the introduction of unicode however, most characters previously considered uncommon were unified in one encoding supported by all browsers, making the Accept-Charset header obsolete [Moz21].

Obviously the introduction of a common standard that makes customization on the server's side unnecessary is not a procedure that is limited to charsets. One could imagine hosts sending content to browsers in every available language instead of listening to the Accept-Language header. But there are also obvious drawbacks to this concerning file size, and with some javascript, the user's choice of language could still be easily extracted from the browser, for example by measuring the displayed length of text sections.

2.4 Javascript

Any website can declare javascript scripts as part of its content so that a visiting browser installation will download and execute them. Among other things, these scripts can make arbitrary computations, manipulate the website they are part of and exchange data with hosts. To enable scripts to do all of these things effectively and efficiently, browsers expose many APIs for purposes such as website manipulation and media rendering, and also provide information about the browser installation's properties, capabilities and the script's execution environment.

All of these APIs and many more can be used as fingerprintable surfaces. Usually, this

1 The rest of the user agent string exists mostly for historic reasons and to signal compatibility with other browsers and rendering engines by impersonating them.

is achieved by interacting with them in a predefined manner while logging their behavior and/or outputs. The logs generated in this way are fingerprints, and the specification of this interaction and logging sequence is a *fingerprinting method.* Fingerprints generated in a browser installation in this manner are usually sent either plainly or as a hash to a host in their fingerprinting network as an HTTP request.

We will begin with our examination of some of the javascript fingerprinting surfaces by examining some attributes of the `navigator` object that either mirror properties contained in the HTTP headers discussed above or are closely related to them.

Attribute	H	Example Value
appCodeName		`"Mozilla"`
appName		`"Netscape"`
appVersion		`"5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36"`
doNotTrack		`"1"`
language		`"de-DE"`
languages	2.716	`["de-DE", "de", "en-US", "en"]`
platform	1.200	`"Win32"`
product		`"Gecko"`
productSub		`"20030107"`
userAgent	7.150	`"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36"`
vendor		`"Google Inc."`

Table 2.1: Fingerprintable Surfaces of the `navigator` object related to HTTP header values. The distinctiveness H is sourced from [GLB18]

Notably, the Accept-Encoding header lacks an equivalent here because the encoding is only relevant for the communication between browser and server, but loses relevance on the site itself, and some attributes have a slightly different format from their HTTP counterparts, but this is an otherwise acceptable alternative surface for the HTTP headers. While retrieving the same property in multiple ways does not help with distinctiveness, as the combined entropy of two fully correlated properties is equal to the entropy of either property on its own, it does help with resilience. Should a browser block access to a property through one surface, there may be others to fall back on.

A striking feature of these `navigator` attributes is the ease with which distinctive properties are accessed here. Some more fingerprinting surfaces that are just as easily accessed and don't require further explanation are listed in table 2.2.

Object	Attribute / Method	Property	Example Value
navigator	cookieEnabled	are cookies enabled?	TRUE
navigator	hardwareConcurrency	available logical cores	8
navigator	maxTouchPoints	max. number of supported simultaneous touchscreen inputs	0
screen	height	screen resolution	1080
screen	width	screen resolution	1920
screen	availHeight	screen resolution	1040
screen	availWidth	screen resolution	1920
screen	colorDepth	color depth of screen	24
screen	pixelDepth	bit depth of screen	24

Table 2.2: Additional simple readouts. The screen resolution surfaces have a combined entropy of $H = 4.847$ [GLB18].

For some of these additional simple readouts, there are again multiple ways of extraction. To give just one example, the `cookieEnabled` property could alternatively be detected by simply trying to set a cookie and reading it again afterwards. We will discuss more alternative methods of extraction in chapter 4.

We will now examine some more fingerprinting surfaces that require a little more explanation than the previous examples.

The plugins list `navigator.plugins` contains an unordered list of all plugins installed on the computer, each represented by an object containing the plugin's name and some other details. Because the list is unordered, it is important to fingerprint it in such a way that the order in which the plugins are read from the attribute does not matter.

The storage APIs `navigator.storage`, `window.indexedDB`, `window.sessionStorage`, and `window.localStorage` can be fingerprinted for existence and accessibility, since some installations do not implement or do not allow access to some of them.

The gamepads function `navigator.getGamepads()` can be used to get a list containing an API object for every gamepad connected to the browser installation's device. These gamepad objects in turn have many properties like `buttons` or `axes` representing their physical properties that can be fingerprinted. Obviously, this surface may be quite unstable depending on how often gamepads are connected to or disconnected from the device.

The speech synthesis API contains `window.speechSynthesis.getVoices()`, which returns a list of all synthetic voices available on the fingerprinted browser installation.

The media query interface `window.matchMedia()` can be used to make queries for information about a browser installation's interfaces. For example, the query `window.matchMedia( '(max-resolution: 300dpi)' ).matches` will be true if and only if the browser installation

is running on a device that has a screen with at most 300 dpi. Using binary search, exact properties can be found in acceptable time. Interesting properties to query include `orientation`, `prefers-color-scheme`, `resolution` and more.

2.5 Membership Query Fingerprints

Because it will come up multiple times in the following sections, we will briefly discuss how lists are fingerprinted if our only access to the list is through membership queries, meaning that we cannot read the full list directly, but we can ask a fingerprintable surface questions in the form of 'is x a member of the list?'.

If this is the case, we can simply choose a subset of the set of all elements that may possibly be contained in the list and query for membership of all elements in that subset. By using the same subset across all fingerprinted installations, we get comparable results, and by choosing a suitably large subset, we may capture most, if not all, of the list's distinctiveness.

For example, a browser installation may have a list of supported image types, but it may not have any simple way to access that list. We can simply embed an image of some type on our website and check if it is displayed correctly by checking the computed size of the `<img>` tag, thus querying the installation for whether that image type is supported. If we could do this for all image types in existence, we would be querying the full list of supported types in this manner, but even querying a small subset of image types may already provide much of the distinctiveness of the full list.

2.6 Media APIs

Modern browsers have capabilities to display not only text, but also media like video, audio, or even 2D and 3D scenes for animation. In order to create media in the browser, they provide mainly three APIs. For visuals, specifically images and animation, websites can use the html canvas object and its associated javascript API. For audio, they can use the entirely javascript-based `audioContext` object. And finally, for video, browsers provide the html video element. All three interfaces make heavy use of hardware acceleration provided by a device's graphics and audio components in order to work efficiently.

The outcome of any operation which uses hardware acceleration may be influenced by the capabilities and implementation of the hardware, its drivers, the operating system and of course the implementation of the javascript API itself. Therefore, differences in these four components between browser implementations may present themselves in media created with the aforementioned APIs, making the rendered media a fingerprint of the components.

Of course, differences in implementations of the four components may be present in places only used by a subsection of one of the APIs. Therefore, we need to use as many of the API's capabilities as possible to capture the maximum number of differentiating properties.

Before we explain how rendering media for fingerprinting works for each of the APIs, we should first mention, that much can be determined about a browser implementation's media capabilities through `navigator.mediaCapabilities`. This API can be asked about the efficiency of a media encoding or decoding scheme and it will answer based on the soft- and hardware capabilities of the browser installation. Answers consist of the three booleans `supported`, `smooth` and `powerEfficient`, indicating whether a given format will likely have the named property on the running browser installation. Because there must be lists of supported, smooth and powerEfficient encodings for every installation, this can be used as a membership query fingerprint.

2.6.1 Canvas

Canvases are elements that allow websites to draw images, either directly or by rendering 2D or 3D scenes. Canvas fingerprinting is done by creating an html canvas element, drawing on it and then extracting the rendered image using `canvas.toDataURL()` or `canvas.toBlob()`. As discussed, it is advantageous for a fingerprinter to use many different capabilities of the API. For canvases, that is usually achieved by drawing many different things before rendering all of them. This can include text, gradients, blending and everything else a canvas can do. Canvas fingerprinting was found to have a distinctiveness of $H = 8.546$ by [GLB18].

WebGL Webgl is an extension of the canvas element that provides a full 3D graphics library. The starting point for using webgl is always to create a canvas element and to access its webgl context with `canvas.getContext("webgl")`. This context can then be populated with 3D objects and used to render images into the canvas with custom shaders. It has multiple fingerprintable surfaces.

Firstly, the webgl context object contains fingerprintable constants like `context.VENDOR` ($H = 2.282$) or `context.RENDERER` ($H = 5.541$), both of which provide information about the GPU encoded as integers [GLB18].

Secondly, webgl canvases can be fingerprinted in the same way as regular canvases, except that instead of simply drawing on the canvas, a 3D scene is populated and subsequently rendered using custom shaders.

Thirdly, there are fingerprintable webgl properties accessible with `canvas.getParameter()`, for example `canvas.getParameter(canvas.SHADING_LANGUAGE_VERSION)`.

Lastly, parameters describing supported precision formats for shaders are accessible through `canvas.getShaderPrecisionFormat()`, for example `canvas.getShaderPrecisionFormat( canvas.VERTEX_SHADER, canvas.LOW_INT)`.

2.6.2 Audio

Websites can use the `window.OfflineAudioContext` API to create sounds. Fingerprinting this API works by creating an audio context, adding a basic sound generator or a sound file to it, applying some effects to the sound and rendering it. For example, `context.createOscillator()` creates a basic sinus wave, and `context.createDelay()` adds an echo to it. Both components need to be configured with `setAudioParam`, for example `setAudioParam(context, oscillator.frequency, 5000)`, and the components need to be plugged together before rendering with `renderAudio(context)`. Again, applying more generators and effects may cover more differentiating properties.

The html audio element can also be used to find out if a browser installation supports an audio encoding, for example with `HTMLAudioElement.canPlayType("audio/mp3")`. This can be used as a membership query fingerprint.

2.6.3 Video

The html video element and its corresponding API do not currently have any rendering or video manipulation capabilities other than `HTMLVideoElement.msInsertVideoEffect()`, `HTMLVideoElement.msHorizontalMirror` and `HTMLVideoElement.msZoom()`, all of which are specific to InternetExplorer and Edge. Frames can be extracted from a video using `CanvasRenderingContext2D.drawImage(videoElement)`, making fingerprinting possible through `canvas.toDataURL()`. It should be noted however, that we did not find any actual examples of video fingerprinting while examining real world fingerprinting scripts. Perhaps the rather complicated implementation and poor browser support makes this not worthwhile in actual applications.

The video element can also be used to find out if a browser installation supports a video encoding, for example with `HTMLVideoElement.canPlayType("video/mp4")`. This can be used as a membership query fingerprint.

2.7 Fonts

When a website requests its text contents to be displayed in a certain font, the browser needs to know how to draw that font. This information can be found in font definition files which

the browser looks for in three different places. First, there are the generic fonts serif, sans-serif, monospace, cursive and fantasy, that come preinstalled with every browser. Second, a website may supply its own fonts through the css `@font-face` rule. Third, the browser looks at fonts installed on the machine it is running on. If a font can not be found on the machine, generally one of the generic fonts is used by the browser as a fallback, but websites can specify a series of fallbacks as well.

From the perspective of fingerprinting, the first option is uninteresting because it won't provide any distinctiveness and the second is uninteresting because it's independent of the browser installation, but the third option is quite useful. Because any program installed on a machine may come with additional fonts, the list of all installed fonts is partially dependent on the list of all installed programs.

Because fonts may display characters at different widths, we can test if a font is accessible to a browser installation in the following way.

Let F be a set of fonts. Let $f \in F$ be a non-generic font. Let $L(t(f)), f \in F$ be the length of a span tag t containing an arbitrary but fixed test string S when it is displayed using font f. We will choose two generic fonts $f_{g1}, f_{g2} \in F$ such that they do not render our test string with the same length: $L(t(f_{g1})) \neq L(t(f_{g2}))$. Should no two such fonts be found, then perhaps the test string should be changed to include more of the characters either font is capable of displaying.

We will create two span tags a and b containing the teststring S. a uses the font f, with f_{g1} as a fallback, while b uses the font f with f_{g2} as a fallback. If f is not accessible to the browser installation, the fallbacks will be used and $L(a) = L(a(f_{g1})) \neq L(b(f_g 1)) = L(b)$. However, if the font is accessible to the browser installation, then $L(a) = L(a(f)) = L(b(f)) = L(b)$.

This method can be used for membership query fingerprinting.

3 Google's Privacy Sandbox

Recall that fingerprinting is only effective if a fingerprinting network hat access to fingerprintable surfaces that provide sufficient amounts of distinctiveness to track a large proportion of visitors while also having small changeability. This observation yields three ways to mitigate fingerprinting:

1. Decreasing distinctiveness of accessible surfaces, for example by making browser installations appear more uniform.
2. Increasing changeability, for example by randomizing surfaces.
3. Decreasing distinctiveness by limiting access to surfaces.

All three are faced with the same challenge. The information and functionality of fingerprintable surfaces is very useful and in some cases necessary for websites to function, and can therefore not easily be modified to be fingerprint resistant in any of the aforementioned ways.

In this work, we focus on the third option. Limiting access to fingerprintable surfaces can be done in different ways. The most obvious would be to outright remove a surface, as was practically done with the Accept-Charset HTTP header. However, we've already explored in section 2.3 why this may not be applicable for most surfaces. Another way is to require explicit user consent before allowing a website to access a surface. This is currently the case with the `geolocation` API and for accessing media devices like cameras and microphones. The consent request interrupts the user, who will likely get suspicious of a fingerprinting website requesting access to multiple APIs seemingly without reason, and can deny the requests or leave the site. A soft way of limiting access was proposed by [Eck10]. It tries to limit font fingerprinting by increasing the time to load a font exponentially with the number of fonts a website already used. This principle could trivially be extended to all membership query fingerprints.

In late 2019, a new method of surface access limitation dubbed the "Privacy Budget" was proposed by the developers of the Google Chrome browser as part of their "Privacy sandbox" project [Sch19b][Sch19a]. With this method, a browser would monitor how much identifying information a website and its hosts have extracted from a browser installation [Sch19b]. Once a

certain, currently undefined threshold of information (the *Privacy budget*) is reached, the website and its hosts would be cut off from accessing surfaces that would reveal more information about the installation.

3.1 Estimating Information

As per the above plan, the Privacy Budget relies on knowing how much information an installation has revealed about itself to a fingerprinter. As we already know, the identifying information revealed by a fingerprinting surface can be calculated with eq. (2.1), but this calculation relies on the probability density function of the surface's obtainable values. To calculate this function, Google proposes to have browser installations actively report the information obtainable through their fingerprintable surfaces to some central collector hosted by the browser vendor [Las20].

It should be noted that the resulting probability density function cannot be used to calculate exactly how much information is revealed through a surface by a browser installation to a particular website and its hosts, because they would have a different distribution of values based only on those browser installations that visit their fingerprinting network. However, if the set of visitors to a fingerprinting network are a sufficiently large random sample from all browser instances, the law of averages tells us that the calculated probabilities are a good estimate. If the selection of fingerprinted browser installations either on the browser vendor's side or on the fingerprinting network's side is skewed in some way, the quality of the estimation decreases.

If Google, or any other browser vendor who wants to implement this proposal only gathers information about the distribution of fingerprints only from installations of their own browser, they will of course get a skewed distribution and won't be able to properly judge the information revealed by their installation. This will be particularly problematic for less popular browsers, because they reveal much information just by the type of browser alone, a fact which would not show up in their collected data, as the type of browser would be uniformly theirs. Therefore, collected data should be exchanged between vendors for the Privacy Budget to work more accurately.

3.2 Privacy Budget

The Privacy sandbox proposal also mentions that browser installations should actively report how much information is extracted from them by visited websites and hosts [Las20]. This

information would then be used to find a sensible threshold for the Budget according to the following considerations [Las20].

By lowering the budget, it would be increasingly improbable for a fingerprinting network to identify a browser installation in a large set of visitors. It would therefore be advantageous to choose the threshold as low as possible. However, as we discussed in section 2.2, it is strictly necessary for many sites to access many fingerprintable surfaces and perhaps inadvertently collect information in the process. In order to not break non-fingerprinting websites, this inadvertent collection should either be identified as not fingerprinting related and not counted towards the budget or be anticipated by increasing the budget. Unfortunately, it is rather difficult to determine whether an access to a fingerprintable surface is fingerprinting related or not, which we discuss further in chapter 4, so the second option of expanding the budget is much more likely to be used. Currently, no value has been proposed for the budget.

Even if the budget is relatively large, some non-fingerprinting websites will still fall outside of it, in which case Google proposes to contact them to see if they can limit their use of fingerprintable surfaces somehow [Las20]. Should this be unacceptable for a website, it could request an exemption from the Privacy Budget rules from the user, who would therein be informed of potential fingerprinting threats [Sch19b][Las20].

One thing that is currently unclear from reading the proposal, is whether Google plans to have a surface's revealed information I or the accessed surface's entropy H count against the budget. There are cases to be made for both. If H is counted towards the budget and a surface s of some browser has $I_s > H_s$, then this surface may reveal more information than the Budget detects, potentially making it uniquely identifiable in a large set. However, if I is counted towards the budget, interaction with a single surface may already exhaust it in some browser installations, and reveal almost no information in others, making it very hard for non-fingerprinting websites to determine how many surfaces they can safely assume to be allowed to use. A balance must be struck here between websites continuing to function and protecting users' privacy.

Another major problem of the Privacy Budget is that it sets a moving target. Even if a website owner designs his site to stay within the budget limitations, changes to the browser landscape and thus the entropy gained from any one surface may mean that the very same website may slip outside of the budget again. This may lead to a drop in acceptance for the Budget both among users and website owners.

3.3 Involuntary access

Currently, much identifying information is revealed to websites and hosts even if they did not request it, and this information would still be counted towards the budget. In order to give websites control of how they want to use their budget, instances of such involuntary access should be reduced.

A prime example of involuntary access are the HTTP headers. Under the proposal, starting at some future version of Chrome, the user agent string will be set to some common, frozen value, and fractions of the information formerly contained in the string called 'user agent client hints' will instead be retrievable at a new javascript API or be optionally sent along with HTTP requests as new headers when needed [TWw21]. Need for a client hint is signaled by an appropriate value in the the Accept-CH header in a server's HTTP response [TWw21].

The proposal also includes removing the Accept-Languages header in favor of a Lang-CH client hint.

4 Detecting Fingerprinting

As we began to discuss in section 3.2, the Privacy Budget cannot be set so low as to break large parts of the web that do not fingerprint. Websites requiring access to surfaces beyond the budget for non-fingerprinting purposes would likely request from the user to be exempt from the Budget rules. If many non-fingerprinting websites did this, the user may get accustomed to granting these exemptions, forming a habit which fingerprinting websites may abuse, similar to how users have been conditioned to accept cookies in areas where the law demands websites to ask for consent before setting cookies. Websites may even shut users out, lest the users grant an exception from the Budget, similar to how many websites react to users who block advertisements from being shown. And if such exemptions are not allowed for websites, too many broken sites would likely prompt users to outright disable the Budget or switch browsers.

In order to estimate how low the budget could be set under these considerations, we must know how much information non-fingerprinting sites inadvertently collect. To measure how much information websites usually collect from fingerprinting surfaces, we visited a large set of websites with a modified browser in which we implemented a system to record websites' accesses to fingerprinting surfaces which we dubbed *FPLog*. We will explain our concrete implementation in chapter 5 alongside our selection of websites, but begin here by explaining how fingerprinting surface access can be detected in theory. This theory also applies to any fingerprinting access detection implemented as part of the Privacy Sandbox.

4.1 Detecting Access

To detect access to surfaces for the purpose of fingerprinting, a browser must first be able to detect access for any purpose. In case of the Privacy Budget, the amount of extracted information must also be determined.

As an example, we will examine the fingerprintable property 'number of available hardware threads'. If it is accessed through `navigator.hardwareConcurrency`, noticing the access is a simple matter of modifying the method in the browser responsible for supplying the value to this attribute, and measuring the amount of extracted information can be achieved by calculating I, assuming we know this surface's probability distribution.

However, the same property may also be extracted by launching a series of truly concurrent web workers until their collective performance begins to drop. In this case, there is no obvious point at which the browser should assume that the property has been accessed. One could try to make the browser notice when the worker's performance is being measured, which is practically impossible considering the countless number of ways in which such a measurement could be made. So in any practical application, the assumption of access may have to be made as soon as even one web workers are launched, considering that the knowledge that an installation has two or more hardware threads may yield distinctiveness already.

Another example of a difficult to detect fingerprinting access is testing the availability of APIs and other functionalities and the presence of bugs to determine a browser's type and version. Because every surface in any browser was surely unavailable or bugged at some point in the browser's history, any access to any surface would have to be considered as extracting version and type information. This is even true for two consecutive accesses to the same surface with the same parameters, as side effects from other changes on the website may have provoked a bug in the meantime. Since the value extracted from this surface depends on whether the exact situation that provokes a bug in some versions of some browsers is present, the Privacy sandbox would need to know about all of these exact situations for an accurate measurement of extracted information. This seems fairly impractical, so it may be necessary to simply always assume that the browser's type and version have been extracted, which immediately uses up part of the website's budget.

4.1.1 Media APIs

Just as we discussed for the number of available hardware threads, many of the ways in which media APIs are used for fingerprinting are also hard to detect because there are multiple ways of extracting the same property. But the media APIs also face another, unique challenge. Because the amount of information extracted from them through rendering media is dependent on what exactly is being rendered, it is rather difficult to determine just how much information was extracted in a surface access, since there is no practical way the browser vendors could get the probability distributions for all media that may possibly be rendered through these APIs.

Therefore, the Privacy Sandbox may have to simply assume their value of H to have been extracted during any render, regardless of what was actually rendered.

4.1.2 Fonts

Font fingerprinting, as well as any other membership query fingerprint, is dependent on the lists of values being queried for. However, every font requested from a browser may deliver a

small amount of information, so it may be useful here to just count the availability of each font as an individual surface. That would also mean, that even just using some basic fonts without going through any long list would be counted towards the budget. Even if we found that fingerprinting networks currently typically requested more than *n* fonts, while most non-fingerprinting sites request less, implementing some threshold underneath which we assume no fingerprinting to have taken place would surely be fully used by fingerprinters to extract information without being limited by the budget anyways.

4.2 Distinguishing fingerprinting and non-fingerprinting access

Any information extracted from a browser can only be determined to be non-fingerprinting related until it leaves the browser's vision. At that point, in order to protect the user's privacy, the Sandbox would simply have to assume that the information is being used for fingerprinting.

The complement is also true: if extracted information, even if it is definitely fingerprinting related, does not leave the browser and reach its fingerprinting network in some way, it cannot actually have been utilized for identification purposes, and is therefore irrelevant.

Therefore, if and only if information extracted from a fingerprinting surface leaves the browser should it be counted towards the budget.

4.3 Traceability of gathered Information

Unfortunately, tracking information as it moves through the browser is simply impractical. Imagine that a website read a value from some surface. One could simply assume that any variable who's value was calculated using the surface's value carries at least part of the surface's value as well. If any such variable leaves the control of the browser, the information contained in the surfaces' value is assumed to have been used for fingerprinting. However, 'calculation' of course also includes any control flow, so as soon as a surface's value or a variable carrying part of it has been used as a conditional, any variable accessed inside the conditional must also be assumed to carry part of the surface's value. This problem gets worse, if side channels like observing caching behavior or timings is considered. In that case, practically any code executed after the access to a surface must be assumed to carry part of the surface's value. Worse still, semi-permanent storage like first party cookies may carry surfaces' values from one session on a website to the next.

In conclusion, it may be most practical for implementations of the budget to assume any

value extracted from any surface to be present in every variable in session storage until the end of the session, and any writing interaction between session and permanent storage to transport all values present in one's variables to all variables in the other.

4.4 Fingerprinting with multiple sites

If two sites are part of the same fingerprinting network and a browser installation visits one of its sites, that site may use its unused budget to fingerprint some surfaces. If the user then follows a link or redirect to the second site in the network, the link may carry the first site's partial fingerprint as an HTTP GET parameter. The second site may then use its own remaining budget to complete the fingerprint. One practical way to use this would be to have one site split itself across multiple domains, for example a news website which shows a list of previews of available articles on example.com along with links to the full texts on reading-example.com.

Considering our arguments on the traceability of gathered information, any redirect to any website carrying any information, perhaps even encoded in the path, may have to be considered as carrying fingerprinting information. This would mean that the privacy budget is emptied even when navigating across different websites, at least as long as links are followed between them. The Privacy Budget proposal currently has no strategy to deal with this. We believe this to be a major oversight.

A similar challenge is posed by IFrames. The proposal does not currently make it clear, whether the information gathered by a site B embedded in an IFrame on site A should be counted towards the budget of A and vice versa. Theoretically, in the HTTP GET parameters, A can send a UUID when setting up the IFrame, perhaps even encoded in the path, which could then be used by the hosts of both sites to exchange any gathered fingerprinting information. Therefore, both sites' budget should theoretically be unified. This creates a large problem for sites which embed other sites they don't coordinate with. The embedded site may increase it's use of the budget and inadvertently break the embedding site.

5 Implementation

As mentioned in chapter 4, FPLog is system for logging a website's access to certain fingerprinting surfaces.[1] It consists of a modified Firefox browser and a Firefox web extension. In this chapter, we will explain how FPLog was implemented, how we checked for its correct functioning, how we used it, and what challenges we identified for an implementation of the Privacy Budget along the way.

To begin with, an important consideration for FPLog is that we want the browser to appear normal to most visited websites. This means that unless a website is specifically checking for the presence of FPLog, the site should function as it does in any regular browser. This is important because if a website experienced some abnormal behavior and stopped execution of its scripts, we would not get to see all surfaces it would usually utilize.

Our implementation also has to consider the challenges described in chapter 4. We decided to only monitor a select set of surfaces, so FPLog does not detect all extracted fingerprintable properties. We also decided to consider any information extracted from any monitored surface as having been used for fingerprinting regardless of whether it actually left the browser, because tracing that information would have required extensive modifications to our browser's javascript engine. Partial fingerprinting with multiple sites has not been considered by our implementation, as there is currently no pressure for websites to use this technique, making us assume it to be quite rare. Also, according to the considerations in section 3.3, access to surfaces in the protocol stack (e.g. TCP / IP / TLS / HTTPS) is not detected by FPLog.

5.1 Javascript

With the sole exception of fonts, every fingerprintable surface monitored by FPLog is accessed by fingerprinting networks through javascript. A complete list of monitored javascript surfaces can be found in table 5.1. All of these surfaces are implemented in native code, meaning they reflect some function or attribute implemented in the browser's core, outside of javascript or its interpreter.

1 The source code of FPLog can be found alongside installation instructions at [Gre21]

In order to get to the native code implementation of a surface, execution steps through various modules of the browser. First, the javascript interpreter has the javascript control flow enter the called function. If the surface is not a function but an attribute, for example `navigator.userAgent`, that attribute's getter is entered instead. Once inside the function, control flow quickly reaches a special instruction for the interpreter that signals a call to native code. At this point, the javascript execution is halted, and the interpreter's control flow finds and enters the appropriate native function. Specifically, a binding is called, which is essentially an API the core browser provides for interactions with the javascript interpreter. This binding layer already contains caches for some values that don't typically change during execution, such as `navigator.userAgent`. However, if a value has no cache or that cache is invalid, the control flow passes through the binding layer into the actual native functions. These are usually organized in ways that reflect the javascript API they implement, so for example the javascript `navigator` object has an equivalent c++ `mozilla::dom::Navigator` class.

To summarize, FPLog could log access to surfaces at any of the following four points: the called javascript function, the javascript interpreter, the binding layer and the surface's implementation in the browser's core. At first, we implemented logging only in the core, only to find out about the binding layer's caching, which caught many surface accesses before they could reach the core. Next, we attempted to log surface access in the binding layer, which proved effective but also made subsequent development of FPLog prone to failures in Firefox's build system. This was because we had to manually disable parts of the build system that would ordinarily regenerate the binding layer from `.webidl` files to make our modifications persistent. This also made it harder for others to build FPLog to reproduce our results, so we dropped this approach.

FPLog ended up logging javascript surface access from two locations: javascript itself, and the browser's core. As for the core, we created a logging module and compiled it as a library we then used across the browser. Native implementations of surfaces were then simply augmented with a call to a function from this logging module to record any access to the surface. These calls to our library can also carry information about the nature of the access.

On the javascript side, we are able to use javascript's capability to redefine an object's properties (both attributes and methods) using the `defineProperty` function. With this, we swap out the ordinary implementations of surfaces for ones that contain calls to our logging. However, the surfaces we swap in also have to behave normally when viewed from the perspective of a website. FPLogs achieves this in two different ways. If a surface only returns a value that is constant throughout a visit to a website, such as `navigator.userAgent`, FPLog reads and stores the surface's value before replacing the implementation with a logger which then returns the stored value on every subsequent access. All surfaces that are cached in the

binding layer are implemented in this way, so that only surfaces without binding layer caches are logged in the core.

If a surface does not return constant values however, any logger overriding it cannot mimic the surface, since there is no way for the logger to access the overridden property. To get around this problem, we added a mirror for every such surface. The native implementations of these mirrors simply call the native implementations of their regular counterparts. For example, the canvas element's `toDataURL()` method is mirrored by the newly created `toDataURLFPLog()`. Note, that all surfaces currently using this mirror approach could have also been logged in the core, completely eliminating the need for this mirroring. Time constraints prevented us from making the necessary changes. Information on which method of logging is used for each monitored javascript surface can be found in table 5.1.

Object	Attribute / Method	Method	Duplicate API
audioContext	constructor	core	no
canvas	constructor	core	no
canvas	toDataURL()	extension	yes
canvas	toBlob()	extension	yes
canvas	getContext('webgl')	extension	yes
canvas	getContext('webgl-experimental')	extension	yes
document	createEvent('TouchEvent')	extension	yes
navigator	appCodeName	extension	no
navigator	appName	extension	no
navigator	appVersion	extension	no
navigator	buildID	extension	no
navigator	cookieEnabled	extension	no
navigator	doNotTrack	extension	no
navigator	hardwareConcurrency	extension	no
navigator	language	extension	no
navigator	languages	extension	no
navigator	maxTouchPoints	extension	no
navigator	mediaCapabilities	extension	yes
navigator	oscpu	extension	no
navigator	platform	extension	no
navigator	plugins	extension	no
navigator	product	extension	no
navigator	productSub	extension	no

Object	Attribute / Method	Method	Duplicate API
navigator	storage	extension	yes
navigator	userAgent	extension	no
navigator	vendor	extension	no
navigator	vendorSub	extension	no
navigator	cpuClass	extension	no
navigator	getGamepads()	extension	yes
screen	height	extension	no
screen	width	extension	no
screen	availHeight	extension	no
screen	availWidth	extension	no
screen	colorDepth	extension	no
screen	pixelDepth	extension	no
speechSynthesis	getVoices()	extension	yes
window	sessionStorage	core	no
window	localStorage	core	no
window	indexedDB	core	no
window	matchMedia()	extension	yes
body	clientWidth	extension	no
documentElement	clientWidth	extension	no

Table 5.1: All javascript attributes and methods for which FPLog logs access, and the means of logging.

Every part of FPLog on the javascript side is implemented as part of a Firefox web extension. It is configured to be loaded and executed on any visited website before the website itself begins its execution, ensuring that FPLog doesn't miss any surface access. Because web extensions can usually not interact directly with the website's javascript execution environment, the surface replacement code is injected into this execution environment by the extension using the `window.eval()` function. Because websites cannot ordinarily write to files, we extended the `navigator` object with a method `logFPLog()`, which forwards its parameters directly to our logging implementation in the browser core, which in turn writes them to a file.

A log entry generated by FPLog to document a website's access to a javascript surface usually shows only exactly which surface was accessed along with the host information described in section 5.3.

We would like to note that while the original method of logging access in the binding layer

was dropped after a while, we did use it to validate the results gathered by the logging FPLog ended up using. Both methods showed the exact same results on a sample of 100 websites.

5.2 Fonts

Because detecting the presence or absence of a font in a browser installation is as simple as rendering elements with same contents using different fonts and comparing their lengths, and because there are countless indirect methods of measuring an element's length in javascript, we decided not to log whether the length of an element rendered in a certain font was measured and instead only log whether a font was requested by a website at all. Because there are also many different ways in javascript to create an element and have it be displayed in a certain font, we decided to not log requested fonts in javascript, but instead in the layout engine. Here, every displayed element is associated with a node (`nsIFrame`) in a layout tree, and this node carries, among other things, the element's computed style (`nsIFrame.mComputedStyle`), including the fonts the element has been requested to be drawn in. Importantly, this also includes fonts not present in the browser installation. FPLog records requested fonts every time the computed style member of any node in the layout tree is modified, including during the node's construction. For every font-family declaration on any element, the names and order of all fonts appearing in the declaration are logged.

5.3 Hosts

Sites can embed other sites through IFrames. In order to correctly link accessed surfaces to visited sites, FPLog logs not only the URL of the site which accessed a surface alongside the access' description, but also the URL of the site's embedder. If a site is visited directly by the browser and not embedded in an IFrame, both URLs are the same.

5.4 Limitations of FPLog

Because FPLog is built on Firefox, any bugs present in the browser that may prevent the correct functioning of a website could skew our results. FPLog also makes changes to the navigator object and other built-in APIs, which could make some sites behave abnormally if they regularly check for being executed in an irregular browser. Most importantly however, there are many fingerprinting surfaces that FPLog does not currently monitor, or monitors only with very rough granularity. For example the individual surfaces accessed during webgl fingerprinting are not monitored at all, instead only the canvas element's `getContext()`,

`toDataUrl()` and `toBlob()` methods are monitored, with the last two being in a way that does not even distinguish between a regular and a webgl canvas. With enough time, the issues around logging detail could be addressed, but for now it is left for future work.

5.5 Selecting Websites to Log

As we explained in chapter 4, a large number of websites should be visited with FPLog to get an idea of how high the Privacy Budget threshold must be for a regular, non-fingerprinting website to function. We also argued, that user acceptance and conditioning will be major factors in determining the Privacy Budget's success. It therefore stands to reason, that when we randomly choose websites to visit with FPLog, the influence a site would have on the acceptance and conditioning of users with regards to the Budget should be it's weight. We could find no published list of websites with such a metric. Instead, we simply selected websites based on their popularity, which we hope is correlated to the number of people which would be influenced by the website with regards to acceptance and conditioning and the extend of that influence.

For our experiment, we visited the top 10000 most popular websites as determined by the methodology of [Poc+18].[2] However, this list is actually a list of popular domains, not websites.

To filter out domains that don't host a website, we let a script automatically make requests to each domain in the list, requesting location `/` through both HTTP and HTTPS. On an HTTP(S) reply with status code 200 (OK), we assume a website is available at the domain. On a reply with status code 301, 302, 303, 307 or 308, all of which are various types of redirects, we follow the redirect as long as is remains on the same domain and recursively make an HTTP(S) request again. The assumption here is that should a popular website be reached through a redirect from a different domain, the popular website would likely have its own entry with its own domain in the list. Any other status code or a timeout leads to an exclusion from the list. We set the timeout to 30 seconds.

This method produced a new list with 10000 entries. Entry number 10000 in the filtered list has index 12561 in the unfiltered list, indicating that $2561/12561 = 20.4\%$ of the original list did not host a website at location `/`.

Some other biases of the list were left uncorrected. Many sites are not actually popular in the sense that regular users would usually visit them, but instead simply share a domain with a high traffic API endpoint. This is particularly notable with advertisement networks,

2 The list of websites can be downloaded at `https://tranco-list.eu/`. It is updated daily and we use the version published on the 17th of January 2021. Subsequent filtering of the list was performed on the 17th of January 2021 as well.

presumably because their content is embedded on many other websites, but we also saw domains for non-http APIs in the list, for example `ntp.org`.

5.6 Automation

Visiting 10000 websites with a browser requires some automation, which we achieved with the selenium library. It allowed us to start instances of our browser, equip them with the FPLog extension and have them visit the 10000 sites in sequence. For each website, only the path that was answered with a 200 during our website selection process was initially visited, but redirects by the site were followed unconditionally. Every website was given 60 seconds to complete all redirects and fully load. If a website did not complete loading in time, we moved to the next site. If the website did completely load, we waited a further 5 seconds to let the website execute some scripts before moving to the next site in the list.

Not exploring visited sites further than the initial redirect from the / path means that we only got a limited view of what surfaces the sites use. This could have been corrected, for example by following all links on the site that stay on the site's domain perhaps recursively up to a certain recursion depth, but for now that is left for future work.

6 Evaluation

Of the 10000 websites visited, we could not gather data for 274. This was usually because the site failed to finish redirecting and loading within 60 seconds or because our browser ran into some kind of error that resulted in the browser's sole tab or the browser itself crashing. All further statistics are based on visiting the remaining 9726 sites. Due to following html and javascript redirects, the actual number of visited URLs was 10542. HTTP redirects were followed, but not added to the list of visited URLs.

Because sites in IFrames can easily have collected data connected to the site embedding them, for example through UIDs in HTTP GET parameters, every surface access made in an IFrame is treated as if it were made by the embedding site. This rule is applied recursively, so that an access made by a site C which is embedded on a site B which is in turn embedded on a site A is counted as an access made by A. Crucially, an access made by count C is not counted towards the site C, as we would otherwise create a bias against sites embedded in different ways on many websites.

6.1 Fonts

Recall that an element can have a list of fonts associated with it that is ordered from highest (0) to lowest priority. If it is available, the font with priority 0 will be used. Otherwise, the font with the next highest priority (1) is used if available and so forth. Therefore, if we want to examine the minimal list of fonts a website needs to have available in order to display properly, examining only fonts that appear somewhere on the site with priority 0 is sufficient.

Of the 10542 websites visited URLs, 10269 or 97.5% used less than 20 fonts with priority 0 and 9640 or 91.4% used fewer than 10. fig. 6.1 shows the exact distribution of sites across the number of fonts used in this category. When including fonts a site only uses with a priority smaller than 0, the distribution widens as seen in fig. 6.2, but 10256 sites still use fewer than 30 fonts.

Going back to only examining font usage with priority 0, the remaining 273 websites using 20 or more fonts are distributed as shown in fig. 6.3. Many of these sites specifically collect data on font availability, usually through javascript. Purposes for this include, but are not limited

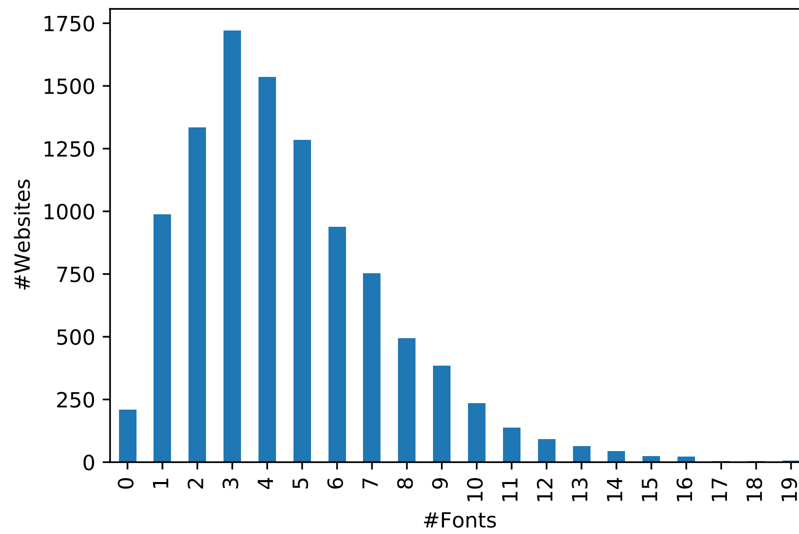

Figure 6.1: Number of websites using between 0 and 19 Fonts with priority 0.

to, font fingerprinting. Some websites also reach these high values because they include many sites through iframes that use different fonts. The highest number of fonts a site used with priority 0 is 640.

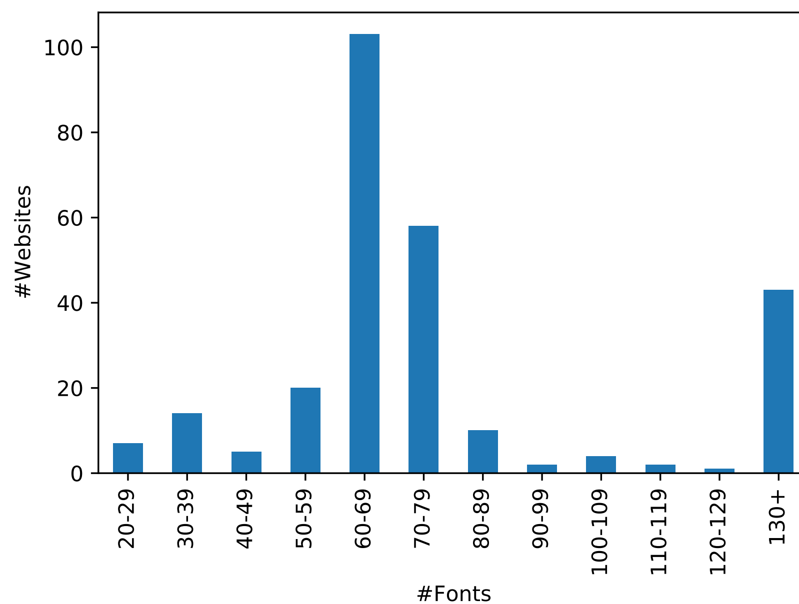

Figure 6.3: Number of websites using between 20 and 640 Fonts with priority 0.

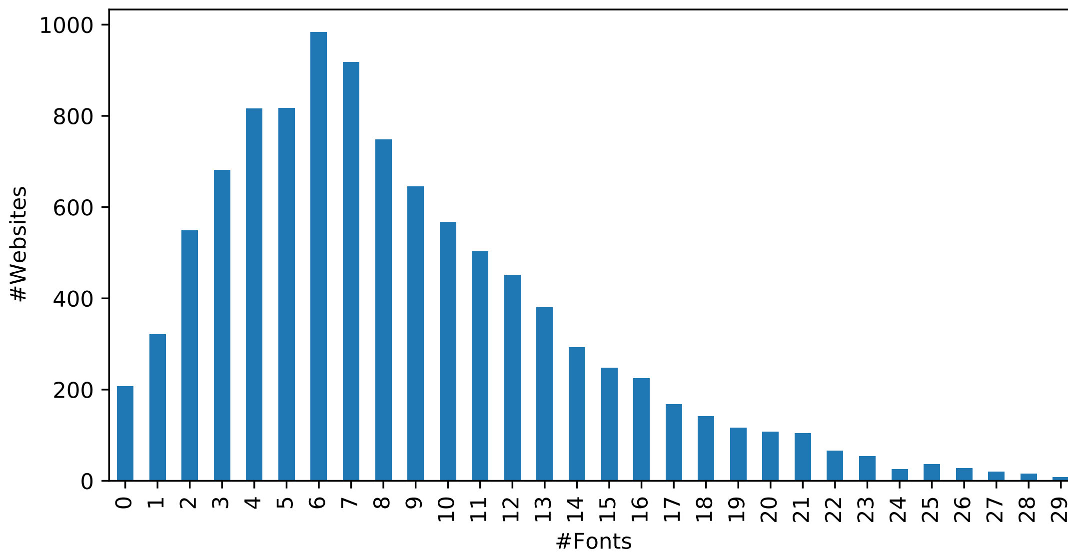

Figure 6.2: Number of websites using between 0 and 29 Fonts with any priority.

We do not have access to information on the distribution of font availability across browser installations, as this data is not currently made available by any research project we are aware of and because we did not have the resources and time to create our own fingerprinting project and collect enough data. Because we only need a rough idea of the minimal necessary Privacy Budget, we will assume that the probability for any font to be available is constant across fonts and that the availability of any one font is statistically independent from the availability of all other fonts. With these assumptions, the 66 fonts that had their ability checked for by the fingerprinting algorithm of [GLB18] would all contribute equally to the 6.904 bits of distinctiveness found for font fingerprinting in that paper. According to this, every font would consume $6.904/66 = 0.1046$ bits from the budget. Looking at fig. 6.2, it seems reasonable to allow for up to 25 fonts (2.615 bits) of any priority in the budget. Considering fig. 6.1 and the availability of generic fonts as fallback, this budget may be tightened to 20 (2.092 bits) and perhaps even 15 (1.569 bits) at some point if enough websites try to reduce their font usage to gain more freedom in other parts of the budget.

6.2 Javascript

The number of websites to access each monitored javascript surface can be found in table 6.1. All of these numbers are out of 10542.

Surface	Parameter	Websites
AudioContext constructor		227
canvas constructor		4320
document.createEvent	touchevent	467
body.clientWidth		6755
documentElement.clientWidth		8824
canvas.getContext	2d	2978
canvas.getContext	bitmaprenderer	177
canvas.getContext	experimental-webgl	445
canvas.getContext	experimental-webgl2	177
canvas.getContext	moz-webgl	22
canvas.getContext	webgl	2159
canvas.getContext	webgl2	205
canvas.getContext	webkit-3d	18
canvas.toDataURL		1648
navigator.appCodeName		732
navigator.appName		4354
navigator.appVersion		7040
navigator.buildId		13
navigator.cookieEnabled		5554
navigator.cpuClass		1101
navigator.doNotTrack		3988
navigator.getGamepads		175
navigator.hardwareConcurrency		1838
navigator.language		8134
navigator.languages		2557
navigator.maxTouchPoints		3155
navigator.mediaCapabilities		474
navigator.oscpu		628
navigator.platform		6062
navigator.plugins		8232
navigator.productSub		903
navigator.storage		375
navigator.userAgent		9980
navigator.vendor		4223

Surface	Parameter	Websites
navigator.vendorSub		584
navigator.webdriver		1978
screen.availHeight		4655
screen.availWidth		4691
screen.colorDepth		8086
screen.height		9123
screen.pixelDepth		1307
screen.width		9155
speechSynthesis.getVoices		451
window.indexedDB		2918
window.localStorage		8414
window.matchMedia		4105
window.sessionStorage		6895

When using selenium to automate a browser, `navigator.webdriver` is true. So the 1978 sites to access this value would have been aware that we are a bot and may have decided to execute differently because of it. This may have biased both our measurements on javascript surfaces and on fonts.

Firefox sometimes calls `navigator.userAgent` when a website accesses some unrelated surface like `navigator.doNotTrack`. We do not now why this happens, but it causes virtually any website to show up as having accessed `navigator.userAgent`, which makes our measurement for this surface meaningless. It is important to keep in mind that all parts of the user agent string can currently still be extracted server-side. When the HTTP user agent header is deprecated and replaced with client hints, client-side access to these surfaces will likely increase.

We do not have access to information on the distribution of surface values or their correlations, meaning that we cannot say how much entropy some combination of surfaces reveals. We can therefore also not calculate a minimum budget. A heuristic we tried using is to group related surfaces like `screen.width` and `screen.height` together and associating the group with a combined entropy which is assumed to have been accessed when a certain number of group members is accessed. Entropy gathered by a website from different groups is then added up as if the groups are completely uncorrelated. Unfortunately, this approach and some others like it involve too much guesswork and for us, they resulted in some websites gathering more than 60 bits of entropy even after double checking many assumptions. This is not compatible

with the results of [Eck10] and [GLB18], both of which found a combined entropy of less than 21 bits for their respective fingerprint datasets. We have therefore simply published our data at `https://github.com/JakobGretenkort/FPLog` and leave it for future work to perform an analysis using actual surface correlation data.

7 Lowering Surface Use

As we mentioned in section 3.2, Google proposes to contact websites which fall outside of the budget in hopes of persuading them to lower their use of fingerprintable surfaces. While we have not done any quantifiable work on this, we did try to find some examples of websites accessing surfaces that could be eliminated.

7.1 Compression

Some websites, for example `https://www.facebook.com/`, use canvases to create images in the user's browser, rather than having the user download them. This could be especially useful for simple animations, as the instructions needed to have a canvas recreate an animation may require significantly less bandwidth than the corresponding video file. It is however often replaceable with an image or a video. Audio interfaces may also be used in this way.

7.2 Compatibility

One thing we have encountered quite often is websites gathering some form of compatibility information, for example for market research or to know what fonts a company should ship with their software. This is a fairly legitimate and fairly widespread use of surfaces and it remains to be seen whether websites will step away from it.

8 Conclusion

In this work, we found some challenges and limitations that will need to be addressed before Google's Privacy Budget or any similar scheme to make limitations to fingerprintable surfaces based on entropy can confidently go forward.

We newly created a clear definition of what fingerprintable surfaces are. In doing this, we also established a new metric for a fingerprint's stability which we managed to relate roughly to distinctiveness. More work has to done on this topic in the future so that we may perhaps find a clear way to calculate the recognizability of a user across multiple visits from only distinctiveness and stability.

In section 3.2, we examined the balance between Privacy and user acceptance that has to be struck when first establishing the Budget. This was elaborated on in chapter 4 and throughout this work. We found that this is one of the major challenges to the establishing of the Budget, with websites having a vested interest to incentivise, condition or force users to grant exceptions from the Budget, similar to how websites currently get users to consent to tracking and disable systems to block websites from showing advertisements.

In section 4.2 and section 4.3, we established that any implementation of the Budget would likely not be able to differentiate whether a surface is accessed for fingerprinting or other purposes and any access would therefore necessarily be assumed to be fingerprinting related.

A major question any implementation of the Privacy Budget needs to address is how to deal with fingerprinting across multiple websites as described in section 4.4. If this problem remains unsolved for either redirects or IFrames, websites may simply circumvent the Privacy Budget entirely.

We were able to implement and use our system for the detection of fingerprinting, FPLog, to gain an insight into how many fingerprintable surfaces websites currently access. However, we were not able to establish a lower bound for the Privacy Budget as we had hoped. For this, we would need data on how fingerprinting surfaces are correlated to one another. Such data is currently not available and both gathering it and combining it with our data on websites' fingerprinting is left for future work.

Both the surface usage we saw in or data from FPLog and what applications we saw using these surfaces on the internet lead us to believe that it is entirely feasible to limit access to

fingerprintable surfaces based on the surfaces' usage of entropy in the manner proposed by Google, and that the limitations could be set up in such a way as to make fingerprinting ineffective for tracking users.

Bibliography

[Ado21] Adobe. *Adobe Flash Player End of Life*. Archived version: https://web.archive.org/web/20210211091223/https://www.adobe.com/products/flashplayer/end-of-life.html. Adobe Systems Software Ireland Limited. Jan. 31, 2021. URL: https://www.adobe.com/products/flashplayer/end-of-life.html (visited on 02/11/2021).

[Bar11] Barth. *HTTP State Management Mechanism*. RFC 6265. Internet Engineering Task Force (IETF), Apr. 2011, pp. 1–37. URL: https://tools.ietf.org/html/rfc6265.

[Eck10] Peter Eckersley. "How unique is your web browser?" In: *International Symposium on Privacy Enhancing Technologies Symposium*. Springer. 2010, pp. 1–18.

[FR14] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. Internet Engineering Task Force (IETF), June 2014, pp. 1–101. URL: https://tools.ietf.org/html/rfc7231.

[GLB18] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. "Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale." In: *Proceedings of the 2018 world wide web conference*. 2018, pp. 309–318.

[Gre21] Jakob Gretenkort. *FPLog*. Karlsruher Institut für Technologie. Mar. 17, 2021. URL: https://github.com/JakobGretenkort/FPLog.

[Las20] Brad Lassey. *Combating Fingerprinting with a Privacy Budget*. Archived version: https://web.archive.org/web/20210301013034/https://github.com/bslassey/privacy-budget. Google LLC. Dec. 14, 2020. URL: https://github.com/bslassey/privacy-budget (visited on 03/01/2021).

[LBM17] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. "FPRandom: Randomizing core browser objects to break advanced device fingerprinting techniques." In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, pp. 97–114.

[Moz21] Mozilla. *Accept-Charset.* Archived version: https://web.archive.org/web/20210123231651/https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Charset. Mozilla. Jan. 9, 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Charset (visited on 02/11/2021).

[Ora21] Oracle. *How do I Enable Java in my Web Browser?* Archived version: https://web.archive.org/web/20210211080723/https://java.com/en/download/help/enable_browser.html. Oracle Corp. Feb. 11, 2021. URL: https://java.com/en/download/help/enable_browser.html (visited on 02/11/2021).

[Poc+18] Victor Le Pochat et al. "Tranco: A research-oriented top sites ranking hardened against manipulation." In: *arXiv preprint arXiv:1806.01156* (2018).

[Sch19a] Justin Schuh. *Building a more private web.* Archived version: https://web.archive.org/web/20210301010111/https://www.blog.google/products/chrome/building-a-more-private-web/. Google LLC. Aug. 22, 2019. URL: https://www.blog.google/products/chrome/building-a-more-private-web/ (visited on 03/01/2021).

[Sch19b] Justin Schuh. *Potential uses for the Privacy Sandbox.* Archived version: https://web.archive.org/web/20210116040800/https://blog.chromium.org/2019/08/potential-uses-for-privacy-sandbox.html. Google LLC. Aug. 22, 2019. URL: https://blog.chromium.org/2019/08/potential-uses-for-privacy-sandbox.html (visited on 03/01/2021).

[TWw21] Mike Taylor, Yoav Weiss, and Mike west. *User-Agent Client Hints.* Archived version: https://web.archive.org/web/20210219013442/https://wicg.github.io/ua-client-hints/. Google LLC. Feb. 23, 2021. URL: https://wicg.github.io/ua-client-hints/ (visited on 03/02/2021).

[Vas+18] Antoine Vastel et al. "Fp-stalker: Tracking browser fingerprint evolutions." In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 728–741.