Fingerprinting Mobile Browsers

Michael Ayele

Helsinki May 22, 2018

Master's thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author
Michael Ayele

Työn nimi — Arbetets titel — Title

Fingerprinting Mobile Browsers

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year May 22, 2018	Sivumäärä — Sidoantal — Number of pages 57 pages + 0 appendix pages

Tiivistelmä — Referat — Abstract

Nowadays, billions of people access the Internet on mobile phones and a significant portion of the traffic comes from browsers. Mobile browsers could be used as a gateway to access the underlying resources of mobile devices for fingerprinting purposes. Browsers include APIs to access the underlying hardware and software resources, such as sensors, audio and media devices, battery, and so on. The growing number of APIs have created new opportunities for browser fingerprinting mechanisms. However, the widely used browser fingerprint systems are designed for the desktop environment and the identifying information gathered using these systems do not include the unique features of mobile phones such as device sensors. The goal of this thesis is to explore additional fingerprintable metrics in the mobile context and analyze their contribution in fingerprinting browsers. In this thesis, we investigated time evolution of browser's features fingerprints and fingerprinting in the wild in the context of mobile devices.

In time evolution of feature's fingerprinting, we have examined the change in permission requirements of browsers over time and evolution of browser's features fingerprints for both Google Chrome and Firefox. In our experiment, we have seen that permission requirements have increased over time, e.g. Firefox 4.0 requires only four permissions, while Firefox 55.0 requires 24 permissions. In evolution of browser's features, we have seen fingerprints that are related to media, audio, WebGL, and canvas elements of the browser show a frequent change across versions. In addition, we have seen, for both Chrome and Firefox, the user agent string is unique for each version and media devices for Chrome is unique for each version as well in our dataset.

In fingerprinting in the wild, we have collected fingerprints from 134 browsing sessions of which 96 were unique. From the gathered dataset, we have calculated the identifying information, entropy, contribution of each browser's feature in our test. The result shows that IP address, user agent, and media devices are the highest entropy contributors. In addition, we have observed that the maximum possible entropy gain in our dataset, 6.58 bits, can be obtained by joining only media devices and user agent strings.

To sum up, in our experiment, we have acquired additional fingerprintable metrics form modern APIs, such as sensors, audio and media devices, and battery. In time evolution of browser feature's fingerprint experiments, we have seen that modern API feature's fingerprints show frequent change across versions. Similarly, in fingerprinting in the wild experiments, these APIs are among the highest entropy contributors.

ACM Computing Classification System (CCS):
Security and privacy → Software and application security → Web application security

Avainsanat — Nyckelord — Keywords
browser fingerprinting, device identification, mobile browser fingerprinting, privacy

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

In networked computing, fingerprinting can be considered as a collection of techniques that collect information about the remote devices for purpose of identification. This gathered information, in turn, is used for creating a unique device fingerprint. In general, fingerprinting methods assume that fingerprints are diverse and stable. The diverse nature of the fingerprints guarantee that no two machine have the same fingerprint and stability requires fingerprints remain the same over time. In fact, diversity and stability are not fully attainable in practice.

The fingerprinting methods can be categorized into two groups based on their data collection modes; namely, passive and active. In the passive group, fingerprinting methods do not directly query the devices for gathering information used for fingerprinting. They are typically based on the subtle difference in client-server request-response communication parameters. For example, wireless device driver fingerprinting [1] and remote physical device fingerprinting with hardware clock skews [2]. In the active group, fingerprinting methods actively query the targeted devices for gathering information used for fingerprinting. For instance, these methods install or run scripts on the device for gathering information from the device and sending that information to servers which create a fingerprint. An example of active fingerprinting is the browser fingerprinting technique of Eckersley *et al.* [3].

Both active and passive fingerprinting techniques can be used in several applications, such as user tracking, second layer authentication, and user experience customization. Internet advertising companies employ device fingerprinting to track users online activities both within a website and across websites [4]. In addition, device fingerprinting can be used in Web authentication. According to Alaca *et al.* [5], they have identified 29 devices fingerprinting mechanism to enhance password based web authentication.

Contrary to its advantages, fingerprinting brings forth challenges to user privacy. Several fingerprinting methods can gather data from features that can be accessed with limited or no user consent; the list of features includes sensor readings, media devices listing, and canvas elements. To mitigate these privacy concerns, there exists a number of privacy protection software applications and guidelines such as the the EFF's Privacy Badger from Panopticlick [3], and the Fingerprinting Guidance for Web Specification Authors from W3C [6].

Browser fingerprinting, a popular fingerprinting technique, can be used in passive and active mode. Passive browser fingerprinting methods rely on the Web requests to gather identifying information about the client device. For example, they can set unique identifiers in HTTP cookies, and extract IP addresses and user agents from HTTP requests. In active mode, the fingerprinting methods implement a website that runs a JavaScript code on the client machine. These scripts extract information related to settings, configurations, and other characteristics of the browser. For example, canvas fingerprinting is based on the subtle differences in the text rendering characteristics of HTML5 canvas element [7]. These fingerprinting methods can be applied both in mobile and desktop environments.

Nowadays, billions of people access the Internet on mobile phones and a significant portion of the traffic comes from mobile browsers. Therefore, the mobile browser can be used as a gateway to mobile devices to access the underlying resources for fingerprinting purposes and then the fingerprints can be used to de-anonymize users on the Internet. However, the widely used browser fingerprint systems are designed for the desktop environment, for example, Panopticlick [3]. Therefore, the identifying information gathered using these systems not fully cover unique features of mobile phones, for example, device sensors. *The main goal of this thesis is to explore additional fingerprintable features in mobile context and evaluate their impact on the mobile browser's fingerprint.*

This work is motivated by the following three works. First, we were motivated by the work of Eckersley *et al.* [3] from 2010. This work provides a very effective fingerprinting algorithm, but it does not cover modern browser APIs. Furthermore, this work it is not aimed at mobile browsers. Second, the recent research work by Al-Fannah *et al.* [8] compares the fingerprintablity of the most popular browsers for both mobile and desktop. This work includes modern APIs and it has covered mobile phones. However, this work does not quantify the impact of the additional features on computed fingerprints. Third, Bojinov *et al.* [9] use device accelerometer readings for fingerprinting devices. Specifically, they observe that the browser sensor APIs can be leveraged by fingerprinting algorithms. *This work extends these works by quantifying the impact of the additional features on computed fingerprints.*

Along with discussing the related work, the thesis work includes four main components: identifying features for fingerprinting, building the fingerprinting tool, analyzing the time evolution of mobile browser's fingerprints, and analyzing mobile browser fingerprints in the wild. Specifically, we extracted 72 features from different sources. These sources include Fingeprintjs2 [10] and DetectRTC [11] libraries, auidocontext fingerprinting code from Princeton CITP's Web Transparency and Accountability Project [12], and our own code. Basically, these features are related to HTTP request, media devices, audio devices, sensors, WebGL, canvas, navigator and windows JavaScript interface of browsers, and miscellaneous fingerprinting techniques. In the fingerprinting tool, we have built a Web application that includes both front- and back-end implementation. The front-end application is used to extract data from user's browser using JavaScript and the main purpose of the back-end application is to extract information from HTTP requests and save this data on a file along with the data that comes from the front-end. In order to examine the time evolution of mobile browser's fingerprints, we have done experiments on the recent 20 versions of Google Chrome and recent 32 versions of Firefox for Android. We observe that the required permissions from browsers increases with each increment in the version number. For instance, Firefox version 4.0 released on 2017-09-28 requested only four permissions while the latest version at the time of our analysis, version 55.0 released on 2013-09-17, requests 24 permissions. In addition, we have tested how features fingerprint changes across versions. For both Chrome and Firefox, the user agent is unique for each version, and media devices for

Chrome is unique for each version as well. In general, media, audio, WebGL, and canvas related features show a frequent change in fingerprint value. In fingerprinting in the wild, we have collected fingerprints from 134 browsing sessions of which 96 were unique. From the collected data, we have calculated identifying information (entropy) gain in bits for each feature and the result shows that IP address, user agent, and media devices are the highest identifying information contributors. In addition, we have observed that the maximum possible entropy gain for our dataset, 6.58 bits, can be obtained by joining only `audio_input_devices` and `user_agent` or `media_devices` and `user_agent` pairs.

The rest of the thesis work organized as follows. Section 2 discusses related work, Section 3 describes the number of features extraction techniques, Section 4 briefly discusses the fingerprinting tool, Section 5 presents the time evolution of mobile browser's fingerprints, Section 6 discusses the fingerprinting in the wild, and finally, Section 7 presents concluding remarks.

2 Related Work

In this section, we briefly discuss the following five related studies: Eckersley [3], Al-Fannah *et al.* [8], Bojinov *et al.* [9], Laperdrix *et al.* [13], and Steven *et al.* [4]. These works are closest to the work presented in this thesis.

The seminal work of Eckersley [3] is the most popular research related to fingerprinting of browsers. Basically, the study examines vulnerability of the Web browsers to fingerprinting. The research includes fingerprinting methods that exploit the version and configuration information of the Web browsers. In the research, data have been gathered from users using a website called `https://panopticlick.eff.org/` and it is used to create unique identifiers. Eckersley observes that browser fingerprinting is very effective way to identify a user without a need to store an identifier on the user machine. For example, in the experiment that observed by Panopticlick, 83.6% of browsers shows unique fingerprints in a particular sample. However, this work is dated because it was done in 2010 and a lot has changed since then. For instance, Web browsers have added several new features, such as sensors, WebRTC, WebAudio, and MediaDevices. Therefore, this work needs to be updated to include newer browser versions. In addition, some of features are specific to desktop environments, thus they may not be used for mobile devices fingerprinting. For instance, mobile phone's browsers do not support plugins. Furthermore, the author points out that the gathered data is biased toward privacy-conscious and technically educated users, and therefore the result may not represent the general public.

Similar to Eckersley's work, Laperdrix *et al.* [13] gather large sample of browser fingerprints (118,934). In this research, we have seen several strengths. The research includes reasonably good number of attributes (17). In addition, it is quite recent work (2016) and it can be used in mobile and desktop environments. Most importantly, it proves that canvas fingerprint is most effective. However, like Eckersley's work, this research lacks modern browser APIs support, such as, sensors, WebRTC, WebAudio, and MediaDevices.

Like Eckersley and Laperdrix *et al.* work, Bojinov *et al.* gather large sample of browser fingerprints (10,000) [9]. It uses mobile accelerometer calibration imperfections and frequency response of speaker-phone and microphone system for fingerprinting. Bojinov *et al.* prove that fingerprinting via mobile sensors may survive a device reset and the entropy gathered form these sensors is enough to uniquely identify a device in amidst thousands of devices.

Unlike the other research works mentioned above, Al-Fannah *et al.* do not collect data from the users. They have examined and compared the fingerprintablity of widely-used browsers for both mobile and desktop and present relative fingerprintablity [8]. The result of the research have shown that Safari is the least fingerprintable, whereas Chrome is the most fingerprintable. In addition, they have observed that WebRTC APIs may reveal some privacy sensitive information. The two main strength of this research project are: it includes modern browser APIs and it covers mobile phone fingerprinting.

Steven *et al.* research work was focused on detecting fingerprinting technique in the wild and they have discovered several new tracking techniques, such as WebRTC, AudioAPI, battery API, and improved canvas fingerprinting. The have designed a software called OpenWPM to crawl websites on the Internet and they have done 15 type of measurements on top 1 million websites. The result of research shows, in addition to new tracking techniques, there is exchanging of tracking data between different sites, e.g. cookie syncing.

3 Mobile Browser Feature Extraction Techniques

In this section, we describe the features and fingerprinting techniques we used.

3.1 HTTP Headers

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems [14]. It is foundation for data communication for the World Wide Web and it defines its request method, error codes and headers. The HTTP headers allow the client and the server to pass additional information with request or the response using name-value pairs. The name of the header is separated from the value by single colon. For example, a request message may contain the following: `accept_encoding: "gzip, deflate, br"`. The header's name is `accept_encoding` and its value is `"gzip, deflate, br"`. Web browsers send the HTTP headers to every website they connect and they out of control of the user. These headers reveal several identifying information about the browser. Therefore, HTTP headers can be used to create a browser fingerprint. In this section, we have discussed HTTP headers that are related to web browser fingerprinting.

accept-documents: It is the accept request-header field that specifies certain media types which are acceptable for the response [14]. In addition, it can be used to specifically limit the desired types by the browser and to indicate preferences. For example, `accept: "text/html,application/xhtml+xml,application/xml;q=0.9, image/webp,image/apng,*/*;q=0.8"`. This line of the request message is interpreted as, the browser preferred media types are `text/html`, `application/xhtml+xml`, `image/webp`, and `image/apng`, but if these do exist, send `application/xml` and if this does not exist send any media type, `*/*`. Note that the *q* parameter indicates the relative degree of preference for that media range, using the *q* value scale from 0 to 1. The default *q* value is 1. The accept header field varies from browser to browser. Therefore, this can be used for fingerprinting. For example, for one of the mobile devices we used the Google Chrome browser sent this value, `"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*; q=0.8"`. Similarly, we observed the following value from a device using Firefox: `text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`.

accept-encoding: The accept-encoding field restricts the content-encoding that are acceptable in the response [14]. For example, `accept-encoding: "gzip, deflate"`, indicates to server, the acceptable encoding type are `gzip` and `deflate`. Similar to accept header, the accept encoding field is not same on different browsers. Thus, this difference can be leveraged for fingerprinting. For example, a Google Chrome Android browser's accept-encoding types are `gzip`, `deflate` and `br`, but a Firefox browser's accept-encoding types are `gzip` and `deflate`. In addition, a

SafariiPhone browser's accept-encoding that looks like: `accept-encoding: "br, gzip, deflate"`. Note that the order of the Safari's accept-encoding types different from the Google Chrome one.

accept-language: It is similar to accept-encoding and accept headers, but it restricts the set of natural languages that are preferred as a response to the request and indicates the preferences of users using q values [14] [15]. For example, `accept-language:"en-US,en;q=0.8"` mean the user prefer English USA, if it is not available, the user accepts other types of English. The browsers send this header to every website they connect. These websites can readily get the complete linguistic preferences of the user and they can use it to fingerprint the browser. For example, let see these three accept-languages headers: `"ru,fr;q=0.8,en-US;q=0.6,en;q=0.4, uk;q=0.2,fi;q=0.2"`, `"en-GB,en;q=0.5"`, and `"en-us"`. The first sample comes from Google Chrome running on Android device and it gives very detailed information about the language preferences of the user. The user's first choice is Russian and it he also accepts French, English USA, other types of English, Ukrainian and Finnish. The second one is from a Firefox desktop browser and it advertises its language preference as British English first and second other types English. The third one is Safari on iPhone and it prefers English USA. Hence, the accept-language header can be used to distinguish users.

X-Forwarded-For: The X-Forwarded-For (XFF) header can be used to identify the originating IP address of a client [16, 17]. The general format of this header is: `X-Forwarded-For: client, proxy1, proxy2`. It contains comma separated list of IP addresses. We took only the leftmost value which is the client IP address. Note that, the value of this header can be collected without user's consent, so this can be another way to fingerprint users of browser.

3.2 Navigator Interface and Window object

JavaScript is a scripting language primarily used by browsers for running Web applications [18]. Its popularity and wide usage makes it easier for browser fingerprinting applications to gather data about web browsers and users. Consequently, most browser fingerprint techniques use JavaScript APIs to access browser features. In this section, we explore JavaScript APIs in the context of browser fingerprinting.

Navigator interface and Window object: Navigator interface and window object are built-in standard global objects. Navigator interface represents the state and the identity of the user agent and it can be accessed through `navigator` interface and the read-only `window.navigator` property. Window object represents the browser's window. These two API are used to get browser specific values and do not require any additional or explicit permissions from the user. Thus, they may be used to gather unique identifying information to create browser fingerprint. Some of these properties have presented as follow:

navigator.userAgent: It returns the current browser user agent string [19, 20]. The user agent string contains several pieces. Some of these pieces are retrieved from other navigator properties, such as `appCodeName`, `appVersion`, `product`, and `productSub`. For example, `"Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5 Build/M4B30Z) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.98 Mobile Safari/537.36"` is a user agent string for Android Chrome browser. This user agent string can be decomposed into the following pieces: a) 'Mozilla/5.0' represents the `appCodeName` and `appVersion`, b) 'Linux; Android 6.0.1; Nexus 5 Build/M4B30Z' represents the platform, c) 'Chrome/61.0.3163.98' represents the name of the application and its version, and d) 'AppleWebKit/537.36 (KHTML, like Gecko)' and 'Safari/537.36' are added just for browser compatibility purposes. These pieces can be used to browser identification of the current browser. However, extracting this information solely from the user agent is not reliable because subsequent versions of browsers can change this information, and users have the freedom to change the user agent string via settings. For instance, there are browser extension for user agent spoofing [21].

navigator.appName: It returns the name of the browser, but because of the backward compatibility, the most browsers return the string "Netscape". Since this property is kept for compatibility reasons, this value may not be useful for fingerprinting.

NavigatorID.appCodeName: It returns the code name of the browser, but because of the backward compatibility, all browsers return 'Mozilla'. Since this property is kept for compatibility reasons, this value may not be useful for fingerprinting.

navigator.appVersion: It returns the version of the browser. It returns either the string "4.0" or a string representing the version of the browser in detail. For example, "5.0 (Linux; Android 6.0.1; Nexus 5 Build/M4B30Z) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.98 Mobile Safari/537.36". Gecko and WebKit based browsers returns "5.0" followed by platform information. This property does not always give correct value, so it is not a reliable source for browser fingerprinting.

navigator.platform: It returns the platform of the browser. It returns the empty string or the browser host device platform detail. For example, "iPhone", "Linux armv7l", "Linux x86_64".

navigator.product: It returns the product name of the current browser, but because of the backward compatibility, most browsers return "Gecko". Since this property is kept for compatibility reasons, this value may not be the correct value.

navigator.productSub: It gives the build number of the current browser [19, 20]. This property is non-standard, thus it does not work on all browsers. For example, Internet Explorer returns `undefined`; Chrome or WebKit navigator compatibility mode browsers return ‘20030107’; Firefox or Gecko navigator compatibility mode browsers return ‘20100101’.

navigator.vendor: It returns the name of browser vendor. It returns ‘Google Inc.’ for Chrome, empty string for Gecko browsers (Firefox), and ‘Apple Computer, Inc.’ for WebKit browsers (Safari) [19, 20].

navigator.vendorSub: It is a vendor version number and it returns always empty string [19, 20].

navigator.buildID: It is a build identifier of the browser. It is given in the form of YYYYMMDDHHMMSS. For example: the string ‘20171003101214’ means the browser is built (updated) on 03/10/2017 at 01:12:14. The string is the same for browsers that have the same version, but user update their browsers at different time and this string contains different values. Therefore, it can be used as an identifier. Note that, this property is supported only on Firefox browsers [20].

navigator.oscpu: It returns a string that identifies the current operating system and the Central Processing Unit (CPU) [20]. For example, the string "Linux x86_64" means the host operating system is Linux (build for 64-bit version of the x86 instruction set). Unlike Firefox, Internet Explorer uses `navigator.cpuClass` to get class of CPU.

navigator.language: It is most preferred language by the user and it is usually the language of the browser UI. For example, en-US. In addition, it is the first element of the array of languages that are returned by `navigator.language` property [19].

navigator.hardwareConcurrency: It returns potentially available logical processors for the user agent. The logical processor core is not same as physical processor core. A physical core may contain number of logical cores. Basically, the logical core refers to the number of threads that can run at same time without context switch. For example, a dual-core CPU may contain four logical processor cores. Thus, this property can be used to profile the concurrency capacity of the host computer's CPU [19].

navigator.doNotTrack: Do-not-track option allows users to indicate to the Web application that they do not want to be tracked. However, it is out of browser control to check if Web application honor this option. If the user set this property in the browser UI, the browser changes the ‘do-not-track’ header accordingly. This property values can be 1, 0, ‘yes’,‘no’, ‘not specified’, ‘undefined’ and ‘null’ depending on the browser type and version. For example, if an Android Chrome browsers returns 1 or ‘null’, 1 means don-not-track and ‘null’ means unspecified. This property is switched off by default in Chrome, Firefox and Safari. Since majority of users do not change the default setting, in contrary to the tracking preference, the users who set do-not-track preference are becoming easily traceable [20].

navigator.connection: It returns the network information about the browser’s host device, such as maximum downlink speed in Mbps and connection type. A type of connection can have one of the following values: Bluetooth, Cellular, Ethernet, none, WiFi, WiMAX, other and unknown [20].

navigator.maxTouchPoints: It gives the maximum number of simultaneous touch contact that is supported by the current device [20]. For example, if the device `maxTouchPoints` is greater than one, it means the browser supports multi-touch. In the case of our research, we use this property with `ontouchstart` and `TouchEvent` events to detect the existence of a touch screen. `Ontouchstart` is fired when a touch take place on the screen surface and `TouchEvent` event reacts to the state of the contact change on the touch-sensitive surface. For example `[5, true,true]` is a value from our data for an Android device, it implies that the device supports maximum five simultaneous touch points and it listens to `ontouchstart` and `TouchEvent` events. Therefore, we can conclude that this device has touched screen and hence is, probably, a mobile device.

navigator.plugins: It returns an array of installed plugins objects. Enumeration of this array is not allowed on modern browsers. However, navigator.plugins defines a property and two methods; namely, `length` property, `item` and `namedItem` methods, to access the content of the array object. For example, the data can be accessed using array bracket, i.e, `notation(plugins[2])`, `item(index)` and `namedItem("name")` methods. Even if this API restricts the enumeration, this privacy measures can be bypassed by using the accessory methods.

navigator.mimeTypes: Similar to `Navigator.plugins`, it returns an array of `MimeType` objects. Enumeration of the array is not allowed on modern browsers. But `navigator.mimeTypes` defines a property and two methods; namely, length property, item and `namedItem` methods, to access the content of the objects; for example, `item(index)` and `namedItem("name")` methods. Even if this API restricts the enumeration, this privacy measures can be bypassed by using the accessors methods.

window.screen.colorDepth: It returns the color depth of the browser screen. For a given device, it represents the allocated number of bits for colors in a pixel [20, 22]. According to the specification, a user agent should return 24, if the user agent color depth is not known or the due to privacy consideration. However, some implementations return 32. For example, we observed that the Android Chrome browsers gives 32 while the desktop Chrome browser returns 24. In both cases, this property doesn't return the actual value. Therefore, fingerprinting browsers using this property is hard unless we are interested in differentiating the standard implementations from none-standard one. Moreover, `window.screen.pixelDepth` returns the same values as `colorDepth`.

window.screen.width and windows.screen.height: The screen resolution of device can be represented in width and height [20, 22], e.g. `[1680,1050]`. These numbers are in pixels and they represent the potential area of the output device that can be used for a user interface rendering. These values vary from a device to a device and they are not restricted by privacy measures. Therefore, the returned values of these attributes can be used as a part of an identifier of a device.

window.screen.availWidth and window.screen.availHeight: Unlike the window.screen.width and windows.screen.height, these values give the available screen area for UI rendering. Since different operating systems reserves spaces for panels, e.g. Mac Dock and menu bar, these value is less than the actual screen height and width [20, 22]. This subtle difference between the available and the actual size opens opportunity for user tracking software. For example, a menu bar in Windows computer can have different size on different user computers. Therefore, this difference can be a useful addition to user fingerprint.

window.devicePixelRatio: It is a ratio of physical resolution to logical resolution [20, 22]. The physical resolution represents the actual pixel size of the device screen and logical resolution is a CSS pixel size. This device pixel ratio varies between devices. For example, the Nexus 5 Android Firefox browser returns 3, but a Firefox desktop browser reports 1. Because of this variation, this property increases the attack surface for fingerprinting.

dateObj.getTimezoneOffset: This method returns the time zone difference between UTC and the local host system time in minutes [23]. For instance, if the user system is located in UTC+3 time zone, -180 is the time zone offset value. Here the offset value is negative because the time zone is ahead of UTC and if the system is behind UTC, the offset value is positive. This offset values change according to daylight saving settings over a year. Therefore, this property result can be used to group the user into available time zones. For the sake of fingerprinting, it can be additional information along other settings.

window.navigator.cookieEnabled: A cookie is a small piece of data that can be stored on user's Web browsers by a server. Cookies are mainly used for session management, personalization, and tracking [24, 25]. Therefore, browsers allow servers to query the permission status of the cookie storage, in advance, through `window.navigator.cookieEnabled`. This navigator interface property returns true or false. This value may indicate the privacy awareness of the user, because browsers return true by default, but if a user change the setting to be returned false, this may indicate the user's knowledge about violation of privacy using cookies. However, the majority of user do not change this setting, as a result, the privacy aware user may be singled out easily by tracking applications.

window.sessionStorage: This property represents session storage area storage object [26, 27, 28]. The data stored in this area gets removed at the end of the current page session. A page session ends when the window or the tab is closed. This property makes session storage less exposed to cookie-like fingerprinting. In cookie-like fingerprinting, basically, web sites store identifying information in user's browser and retrieve the data later. However, if the browser window is left open for a long period of time, the user can be exposed for fingerprinting in that time frame.

window.localStorage: Unlike to session storage, local storage keeps the stored data across browser sessions. Therefore, this local storage's data persistence property makes the browser vulnerable to cookie-like fingerprinting unless the browser clears the data for security reasons [29, 30].

window.indexedDB: Similar to local storage, indexed database is persistent data store. `IndexedDB` is a key-value in-browser database and it provides an advanced query API for data processing. As local storage, `indexedDB` exposed to cookie-like fingerprinting. For example, if a third-party tracker is able to create a unique identifier and store it in the `indexedDB`, the user can be tracked across multiple sessions [31, 32].

window.openDatabase: It is a in-browser Web SQL database [33]. The database uses SQL for data administration. Like the other browser storage systems, Web SQL database is vulnerable to the cookie-like fingerprinting.

navigator.permissions: This API allows the website to query the current browser permissions status with a unified interface. The state of a permission of a specific feature is either `"granted"`, `"prompt"`, or `"denied"`. The state of permission information helps the website developers to determine whether a permission request is needed for using a specific feature API. For example, `"granted"` infers that there is no need for prompting the user and that permission is already granted, `"denied"` indicates no need for prompting the user and that permission is already denied, and `"prompt"` means user should be asked to resolve the permission state. Currently, this permission API supports a few APIs, such as

Geolocation, Notification, Push and MIDI. In addition to Navigator.permissions, `navigator.getUserMedia` can be used to get additional permission states, such as camera and microphone permissions. Therefore, the permission state of the browser can be used to fingerprint user's preference. For example, a tracker can create an object that represents the user preference like this: `{"webcam":"denied", "microphone":"denied", "geolocation":"denied", "notifications":"granted", "push":"granted", "midi":"denied"}`; 'webcam' and 'microphone' properties have two possible values, granted and denied and the rest properties have three possible values, prompted, granted, and denied. Thus, this object could represent unique $2 \times 2 \times 3 \times 3 \times 3 = 108$ possible values [34, 35].

navigator.getBattery: It is a promise-based API to monitor the battery status of the device. It provides information such as current power level, the charging status, battery charging time, and battery discharging time. Like other Web APIs, the battery status API can be vulnerable to fingerprinting. For example, according to [36], the battery status API can be used to fingerprint a device for short time intervals by analyzing the differences between levels, `chargeTime` and `dischargeTime` [37, 38].

3.3 Browser Tampering

According to the *fingerprint2.js* browser fingerprinting library [10], browser tampering includes range of techniques that are employed to randomize/substitute the native browser libraries values to reduce the uniqueness of the browser. For example, a user can use plugins or a JavaScript code to change the userAgent, screen resolution, or OS name. On the contrary, this browser tempering effort may be source of additional fingerprinting entropy. In this subsection, we will see some of these techniques as presented in `fingerprint2.js` library:

has-lied-languages: In this technique, the library compares the navigator.language value with the first language value of navigator.languages array. Thus, if the comparison result is false, the user tampered the language, because these values are expected to be equal on not tampered browsers.

has-lied-resolution: As has-lied-languages, the library compares `screen.width` and `screen.availWidth` or `screen.height` and `screen.availHeight` properties. Under normal circumstances, `screen.width` is greater than `screen.availWidth` and `screen.height` is greater than `screen.availHeight`, but if the result is otherwise, the user has lied about the resolution.

has-lied-os: In this method `fingerprint2.js` basically compares the OS value that is extracted from userAgent with `navigator.oscpu` or `navigator.platform`.

If the value returned from `navigator.oscpu` or `navigator.platform` is not equal to the one extracted from userAgent, than the user has faked the OS.

has-lied-browser: Similar to has-lied-os, the library compares the browser name that is extracted from userAgent with `navigator.productSub` or `eval.toString().length`. If the value returned from `navigator.productSub` or `eval.toString().length` is not equal to the one extracted from userAgent, than the user has faked the browser.

3.4 Private Browsing and Adblock

is-private-browsing: The private mode browsing is designed to hide browsing activities of the user by disabling the browser history and in browser storage. A remote website can use several techniques to learn if the user is in private mode. For example, a JavaScript library called DetectRTC [11] examines the behaviors of the web storage to detect mode of browsing. For example, under private browsing mode on Firefox, if a website tries to open an `IndexedDB` database, the browser returns error. Therefore, the website can deduce form the errors, the user is in private browsing session.

adblock: It is a set of methods that detects if the user is using an AdBlock plugin using JavaScript. According to `fingerprint2.js` library source code, detecting AdBlocker, basically, involves three main DOM manipulation steps:

1. Create an `adsbox` element (`adsbox` is a removable element class according to AdBlock Plus documentation [39]).
2. Add this element to a document node.
3. Query the 'offsetHeight' of this element and the element will not have any height if an AdBlocker is installed.

Note that this technique does not work for all AdBlockers, for example Ghostery [40]. Since AdBlockers are not used by all users, a browser with AdBlocker are in smaller set among the larger user base. Therefore, the existence of AdBlocker can be source of additional fingerprinting entropy.

3.5 Operating System Name and Version

Operating system name: According to DetectRTC library [11], the operating system name can be extracted from user agent strings of the browser. If the user agent contains either 'iPhone', 'iPad', 'iPod', or 'i' string, The operating system is iOS. Given a user agent, for example: "Mozilla/5.0 (iPhone; CPU iPhone

OS 11_0_3 like Mac OS X) AppleWebKit/604.1.38 (KHTML, like Gecko) Mobile/15A432", the operating system name is iOS.

Operating system version: Similar to operating system name [11], the OS version can be obtained from userAgent and appVersion strings using regular expression matchings. Given a user agent, for example: "Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5 Build/M4B30Z) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.98 Mobile Safari/537.36", the version of the operating system is 6.0.1.

3.6 List of Fonts

JS/CSS fonts: JavaScript code can detect if particular font is available or not in the browser [41]. This method relies on the fact that different font families render a character with different width and height for a same font-size. Basically, the library follows three main steps to obtain the list of supported fonts. First, it creates a string in a generic base font and measures its width. Second, it creates the same string in the font which presence we want to test in the system and measures its width as well. Then, finally, it compares the tested font measurement against the base font measurement and if it is equal, it indicates that the string is rendered using a fallback base font; therefore, the font is unavailable, otherwise the font exists. The test is repeated for an array of exhaustive list of font families to obtain the list of supported fonts. Therefore, because of this list elements and/or order is not the same on different browsers, it can be used as identifying information for fingerprinting.

3.7 Sensors API

Nowadays mobile devices are always equipped with several sensors, such as accelerometer, ambient light sensor, GPS sensor, compass, proximity sensor, pressure sensor, and gyroscope. These sensors are exposed through JavaScript APIs to interact with web applications. Essentially, sensor APIs are a way to present sensor reading to the Web. As other APIs, the Web sensor APIs opens new opportunities for fingerprinting attacks. In this sub section, we will give a general overview of sensor related fingerprinting attacks.

navigator.geolocation: It provides location information for web applications, such as `latitude`, `longitude`, `altitude`, `accuracy`, `altitudeAcuuracy`, `heading`, and `speed`. The latitude and longitude values are in decimal degrees and they indicate the geographic coordinate of the device on Earth. The altitude property represents the height above the Earth ellipsoid of the device in meters. The `accuracy` and `altitudeAcuuracy` specifies the confidence level of the measurements for latitude and longitude coordinate, and altitude respectively. Heading refers to the movement direction of the device in degrees and speed is horizontal velocity of the device in meters per second. Therefore, all these API attributes reveal a range of privacy sensitive information and the data can be used for device fingerprinting purpose [42, 43].

DeviceOrientation Event: This event provides data about the physical position of the device with respect to the local coordinate frame. According to the W3C specification [44], local coordinate frame of a device is defined by three coordinate axes, `x`, `y` and `z`. X-axis is a horizontal line on the plane of the screen from left of screen to the right, y-axis is a vertical line on the plane of the screen from bottom of screen toward up, and z-axis is a perpendicular line to the plane of the screen from inside to the outside. These axes are referenced to the device portrait orientation. Thus, the device orientation event provides orientation of the device with respect to the local coordinate frame in three rotational angles, `alpha`, `beta`, and `gamma`.

- `alpha` measures the device rotation angle around its z-axis within the range of [0, 360).
- `beta` measures the device rotation angle around its x-axis within the range of [-180, 180).
- `gamma` measures the device rotation angle around its y-axis within the range of [-90, 90).

Having enough data gathered form this orientation sensor, for example, the remote website can learn how the user is holding the device. Therefore, the device orientation API can be used for behavioral fingerprinting.

DeviceMotion Event: It provides information about the movement of the device, such as the acceleration and rotation rate of the device with respect to the device coordinate frame. In addition, the event includes metadata, like the event firing rate of the underlaying hardware. Some of the device properties defined as follows [44, 45]:

- `acceleration` is the acceleration of the device in x, y and z axises in meters per second squared.
- `accelerationIncludingGravity` is acceleration of the device including the effect gravity in x, y and z axises in meters per second squared.
- `rotationRate` represents the device's rotational angle change in degree per second.
- `interval` is the motion event firing rate of the device in milliseconds.

The data obtained from this API can be used for fingerprinting purpose. For example, Bojinov *et al.* [9] show how the accelerometer's calibration error can be used for fingerprinting purpose. Furthermore, the event firing rate of the device can be a potential source of identifying information.

DeviceLight Event: It presents the ambient light level of the device in lux [46, 47]. The illuminance level assists website designers to customize the web pages features according to the device environment light level, e.g. change screen's brightness to fit the device's surrounding light level. Even if the API offers many beneficial use cases for web developers, it includes security and privacy risks as well.

This API can be exploited by remote website to profile a user. For example, if the light intensity information is gathered from device over a period of time, it can be used to model environmental use patterns of the user.

Proximity Events: It provides proximity level of a device in centimeters [48, 49]. Basically, proximity level shows how close a user is to the device. Device proximityEvent interface contains three read-only properties: `DeviceProximityEvent.max`, `DeviceProximityEvent.min` and `DeviceProximityEvent.value`.

- `DeviceProximityEvent.max`: maximum sensing distance.
- `DeviceProximityEvent.min` : minimum sensing distance.
- `DeviceProximityEvent.value`: current proximity value.

The max and min value indicates the capacity of the proximity sensor on a device and these values varies between devices. Therefore, this sensing range can be used to profile the device.

3.8 JavaScript Media Related APIs

Similar to sensor APIs, media related APIs can be an attack surface for browser fingerprinting. In this subsection, we have discussed briefly about `mediaDevices`, WebRTC, and `mediaRecorder` API.

media-devices: It provides information about the available input and output media devices, such as microphones, cameras and speakers [8, 50, 51, 52]. Media devices API defines several methods to work with the underlying media devices. For example, `mediaDevices.enumerateDevices` returns `mediaDeviceInfo` object which defines several properties to describe input and output devices. Some of these properties briefly described as follows:

- **MediaDeviceInfo.deviceId**: It is a unique identifier of a connected media device. This value is not the actual model ID of the hardware device, but it is hash of browser's device ID salt, the origin website URL, and the actual physical device ID. This value is persist across browser sessions for months and they are unique for the origin website. Note that since the salt value of the browser changes if the user clear the browser data or use private browsing mode, the device ID value is not persistent in these use cases. However, this property value can be used to profile a user over a period of time if the use does not clear the browser data frequently. In addition, the attacker can leverage `iframes` to embed the same web content on several websites to obtain the same unique identifier across different origins.

- **MediaDeviceInfo.groupId**: It is a session-unique group identifier for the input and output media devices. Media devices in the same group has the same `groupID` and these devices are on the same physical device. This value is not consistent between sessions, so it is not useful for fingerprinting purpose.

- **MediaDeviceInfo.kind**: It returns the kind of the media devices and its value is either "videoinput", "audioinput" or "audiooutput". `Videoinput` string represents video input devices, such as cameras; `audioinput` represents audio input devices, such as microphones; and `audiooutput` string represents audio output devices, such as speakers. The existence of these strings show that the host device holds those kinds of devices. Therefore, a tracking website can easily query the existence of cameras, headphones, speakers and microphones using media devices API property and the returned value can be used for user identification purposes. For example, if the device.kind property value is 'videoinput', the host device has a camera; if the device.kind property value is 'audiooutput', the host device has speakers; and if the device.kind property value is 'audioinput', the host device has a microphone.

- **MediaDeviceInfo.label**: It is a label of the device that is given by the user agent, such as 'Integrated Camera', 'Internal microphone', and 'External USB webcam'. Therefore, this fine grained device information can be used for fingerprinting purposes.

Local IP Address: The user local IP address can be disclosed by using WebRTC APIs. WebRTC is a technology which enables real time peer to peer communication between Web browsers. It defines several APIs for establishing communication channels and exchanging data between peers. However, this API exposes some privacy sensitive information at the same time. For example, an attacker can reveal the user local address using the following steps:

1. The browser sends a request to STUN server.

2. The STUN server sends back a response that contains the IP address.

3. The browser parses the response and sends the IP address to the server.

MediaRecorder.isTypeSupported: MediaRecorder API is designed to facilitate recording of audio and video media streams in Web browsers [53, 54]. It defines several properties and methods for processing and analyzing the captured data. For example, `IsTypeSupported` is used to check if the browser supports the specified codec, container, or MIME type. Therefore, similar to list of pulgins fingerprinting, an attacker can build an array of possible video and audio MIME types and check if the browser supports them. This list of supported media streams MIME-types can be used for fingerprinting the browser.

3.9 Web Audio API

Basically, Web Audio API is designed for processing and synthesizing audio in Web applications. This API provides number of interfaces for Web developers to choose the audio source, apply audio effects, create visualization and so on. As many other Web APIs, Web audio is exposed to security and privacy challenges. In this subsection, we will discuss how an attacker can use we audio API for browser fingerprinting purpose [55, 56, 12, 4].

AudioContext properties: It is a collection of `audioContext` properties for a given browser. These values show the underlying hardware audio stack processing capacity. Therefore, this collection can be used to create unique profile of the browser by either hashing the entire collection to a single string or analyzing the collection object as it is. For example, brief description of some of the collection entries is given as follow:

- **audioCtx.baseLatency**: It is the extra latency time added, in seconds, by `audioContext` processing engine when it transfers the audio object from destination audio context node to audio subsystem of the device.
- **baseAudioContext.sampleRate**: It is audio sampling rate of audio processing nodes in sample per seconds.
- **audioDestinationNode.maxChannelCount**: It is the maximum number of channels that a destination audio device can handle.

OscilatorNode based fingerprinting: It uses `OscilatorNode`, `AnalyzerNode`, `GainNode`, `ScriptProcessorNode` elements of the `AudioContext` object to create a fingerprint. Basically, the fingerprint is a list of the real-time frequency response values for a given audio wave that is generated by `OscilatorNode`. According to the source code obtained at `https://audiofingerprint.openwpm.com/` [12], the algorithm follows these steps:

1. Create an `AudioContext`.

2. Create `OscilatorNode`, `AnalyzerNode`, `GainNode`, `ScriptProcessorNode` from the `AudioContext`.
3. Create an audio wave using `OscilatorNode`, for example, triangle wave.
4. Make the `GainNode` gain 0 to disable the volume.
5. Connect `OscilatorNode` with `AnalyzerNode`, `AnalyzerNode` with `ScriptProcessorNode`, and `ScriptProcessorNode` with `GainNode`.
6. Listen the `ScriptProcessorNode` `onaudioprocess` events.
7. Extract the frequency data from the events.
8. Store them in a data structure.

This list of values may be used as unique identifier because it is gathered from the frequency analyzing process which in turn is dependent on the underlying hardware audio stack.

OscillatorNode and dynamicsCompressorNode based fingerprinting: This fingerprint technique uses the same algorithm as oscillator based one. However, it applies a dynamic compressor to `OscilatorNode` output audio wave to control the signal level and to avoid distortion. Therefore, this method may be more effective than oscillator-only method.

3.10 WebGL API

In this subsection, we will discuss briefly about WebGL fingerprinting. Essentially, WebGL is a JavaScript API that is used for rendering 2D and 3D images in the Web browsers without a need for plugins; and WebGL fingerprinting refers to all browser fingerprinting techniques that are based on this API. First, we will see a technique from `fingerprint2.js` library, and then we will see how the WebGL debug information may be used for fingerprinting.

WebGL image and report: According to the `fingerprint2.js` library documentation [57], an identifying string may be created form a combination of a WebGL image hash and a WebGL report dump. First, it creates an gradient image with a shader and then it converts the image into Base64 string. Second, it lists the WebGL report dump which includes all WebGL capabilities and extensions. Finlay, it concatenate both the base64 string and the report dump to create a huge string. Therefor, this concatenated string can be used to identify a user device because the creation of the image depends on the GPU of the the device and the WebGL dump contains a device specific information such as WebGL vendor and renderer.

WEBGL debug renderer info: It is a debugging extension for WebGL API applications and it reveals information about the graphic driver, namely vendor and renderer string constants [58, 59]. The vendor string constant represent the company who designed the graphic driver, such as Google Inc. and ARM. The renderer string show the details of the graphic driver renderer, for example 'ANGLE (Intel(R) HD Graphics 4000 Direct3D11 vs_5_0 ps_5_0)'. Thus, since the information that is gathered from WebGL debug renderer info extension contains detailed data about the graphic driver, this property can be used for browser fingerprinting purpose.

3.11 JavaScript Canvas API

Primary, the HTML5 Canvas element is designed to create graphics with JavaScript, but it may be used as user tracking method as well. Canvas fingerprinting is a tracking technique that leverages the fact that a canvas image is rendered differently on different devices to build a unique system fingerprint [60]. Basically, this subtle difference in canvas image rendering originates from the fact that the canvas API relays on the underlying hardware and operating system to create images efficiently [61]. As an example, we present the algorithm that is implemented in `fingerprint2.js` library as follow:

1. Create canvas element.
2. Get the canvas context.
3. Fill the canvas context with a test text.
4. Apply canvas winding and blending to enhance the uniqueness the image.
5. Export the canvas image to base64-encoded string by using `toDataURL()` function.
6. Take the exported string as a fingerprint.

According to Mowery *et al.*, canvas fingerprinting technique is "consistent, high-entropy, orthogonal to other fingerprints, transparent to the user, and readily obtainable" [61].

4 Ashara: Fingerprinting Tool

In this section we present Ashara, a tool we have developed to collect fingerprints from web browsers. We named our tool Ashara because Ashara means fingerprint in the Amharic language, Ethiopia. Ashara is Web-based, and like any Web-based tool it has two key components: the front end, and the back end. We now detail these components.

4.1 Front End of Ashara

The front end component is a simple Web application which includes HTML and JavaScript codes. The JavaScript code primarily contains implementations of browser fingerprint algorithms. It contains the code developed during the course of the thesis work, and also from two open source fingerprinting libraries: `fingerprint2.js` [10] and DetectRTC [11], and codes form Princeton WebTAP project [62] for web audio fingerprinting. `Fingerprint2.js` is a popular browser fingerprinting library, which is able to extract more than 25 fingerprinting features such as user agent, screen resolution, etc. We used DetectRTC for extracting fingerprinting features when browsers support WebRTC. The list of WebRTC related features include the number of audio-video devices, existence of speakers, etc. Along with these libraries we added code for extracting features from several other sources such as sensors, media recorders and additional browser's navigator and window object properties. The complete list of features and the corresponding libraries used by Ashara is described in Table 1. The detailed description these features is given in Section 3.

In addition to JavaScript code to extract features, the Ashara's front end also uses jQuery deferred object to sequentially execute asynchronous operations. The list of operations includes requesting permissions and also jQuery's Ajax API to exchange data with the back end. We would like to point out that extracting the features using the Web audio context and Sensors APIs was non-trivial. The Web audio fingerprinting was showing inconsistent results. This inconsistency is largely due to the interference of the remaining source codes which are running at the same time. At the same time, we observed a large variance in the volume of data generated from the sensors such as orientation sensors.

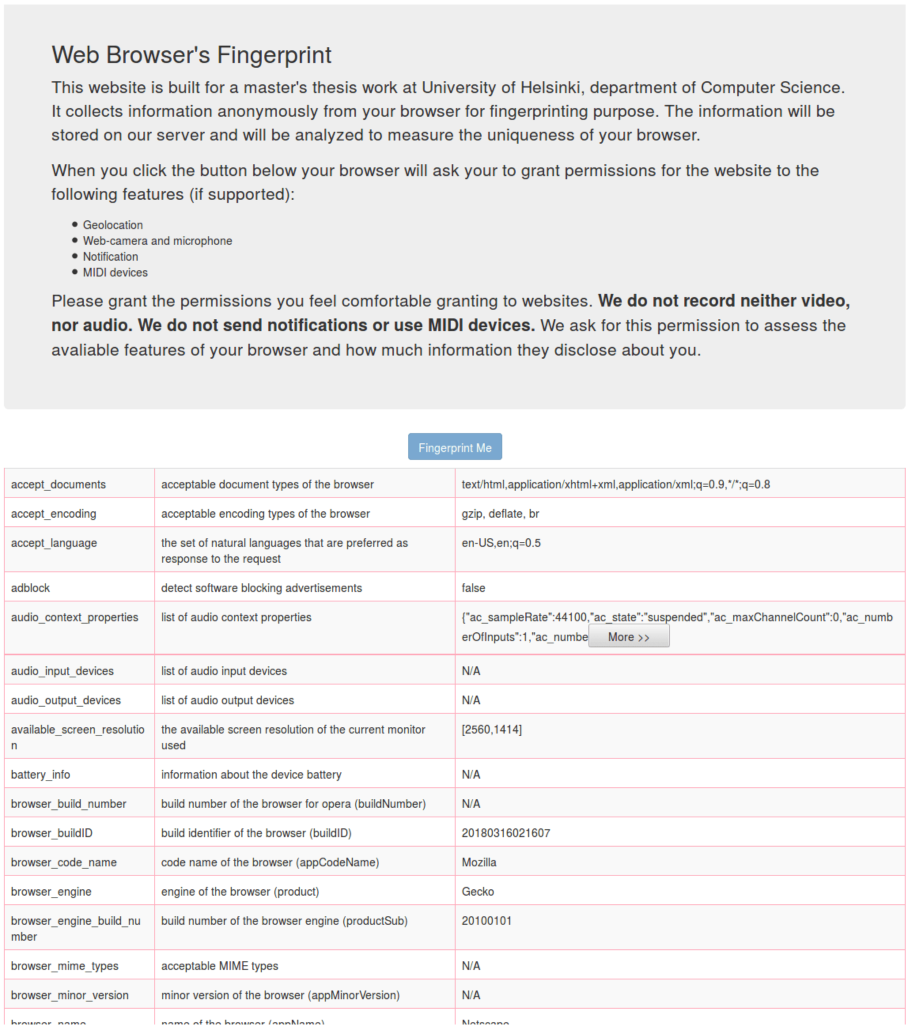
To address these issues, we have divided the execution of the source code into two phases: *Document Load* and *Button Clicked*. During the *Document Load* phase the user will be asked permissions for giving the Ashara site permissions to access sensors. The list of permissions includes location, camera, microphone, notification, and midi.

Feature	**Source**
accept_documents	Our Code
accept_encoding	Our Code
accept_language	Our Code
ip_address_from_server	Our Code
available_screen_resolution	fingerprint2.js
battery_info	Our Code
browser_build_number	Our Code
browser_buildID	Our Code
browser_code_name	Our Code
browser_engine	Our Code
browser_engine_build_number	Our Code
browser_mime_types	Our Code
browser_minor_version	Our Code
browser_name	Our Code
browser_vendor	Our Code
browser_vendor_version	Our Code
browser_version_and_platform	Our Code
browser_version_number	Our Code
color_depth	fingerprint2.js
cookies_enabled	Our Code
cpu_class	fingerprint2.js
do_not_track	fingerprint2.js
hardware_concurrency	fingerprint2.js
indexed_db	fingerprint2.js
open_database	fingerprint2.js
session_storage	fingerprint2.js
local_storage	fingerprint2.js
navigator_platform	fingerprint2.js
network_information	Our Code
operating_system_language	Our Code
os_and_cpu	Our Code
permissions_after_fingerprint	Our Code
permissions_before_fingerprint	Our Code
pixel_ratio	fingerprint2.js
preferred_language	fingerprint2.js
regular_plugins	fingerprint2.js
screen_resolution	fingerprint2.js
touch_support	fingerprint2.js
user_agent	fingerprint2.js
media_devices	Our Code
audio_input_devices	DetectRTC
audio_output_devices	DetectRTC
video_input_devices	DetectRTC
media_recorder_audio_mime_types	Our Code
media_recorder_video_mime_types	Our Code
has_microphone	DetectRTC
has_speakers	DetectRTC
has_web_cam	DetectRTC
ip_address	DetectRTC
audio_context_properties	CITP Project code
web_audio_fingerprint_oscillator	CITP Project code
audio_oscillator_and_dynamicsCompressor	CITP Project code
device_current_position	Our Code
device_light	Our Code
device_motion	Our Code
device_orientation	Our Code
device_proximity	Our Code
has_lied_browser	fingerprint2.js
has_lied_languages	fingerprint2.js
has_lied_os	fingerprint2.js
has_lied_resolution	fingerprint2.js
has_lied_resolution	fingerprint2.js
os_name	DetectRTC
os_version	DetectRTC
canvas_fonts	fingerprint2.js
js_css_fonts	CITP Project code
is_private_browsing	DetectRTC
adblock	DetectRTC
canvas	fingerprint2.js
webgl	fingerprint2.js
webgl_renderer	Our Code
webgl_vendor	Our Code

Table 1: **Sources for the features extracted by Ashara.** *We extract a total of 72 features. Of these features we use fingerprint2.js for extracting 25 features, we use Princeton CITP Project code for extracting 4 features and we use DetectRTC for extracting 10 features. Along with these libraries, we extract an additional 33 features using our own feature extraction code.*

After the permissions have been granted, the code for extraction of Web audio and

Sensors related features is first executed. This allows us to set timeouts to force the program to run feature extraction sequentially. Sequential execution and timeouts are essential because some features require active data exchange with sensors. For instance, basically, AudioContex oscillator fingerprinting involves creating audio wave with oscillator node, send the wave to analyzer node, and receive the frequency response of the analyzer. We did not want such active probing of a sensor to affect the readings of other sensors. For extracting sensor fingerprints, we take the first 500 readings. During the sl Button Clicked phase, the rest of the features are extracted. This phase begins when the user clicks on the button to start the fingerprinting. Note that, although the collection of the data begins after the user grants the permissions to Ashara site, the collected data is not sent to the back end server if the user does not click this button. A screenshot of the content shown to the Ashara's users is shown in Figure 1.



Web Browser's Fingerprint

This website is built for a master's thesis work at University of Helsinki, department of Computer Science. It collects information anonymously from your browser for fingerprinting purpose. The information will be stored on our server and will be analyzed to measure the uniqueness of your browser.

When you click the button below your browser will ask your to grant permissions for the website to the following features (if supported):

- Geolocation
- Web-camera and microphone
- Notification
- MIDI devices

Please grant the permissions you feel comfortable granting to websites. **We do not record neither video, nor audio. We do not send notifications or use MIDI devices.** We ask for this permission to assess the avaliable features of your browser and how much information they disclose about you.

accept_documents	acceptable document types of the browser	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept_encoding	acceptable encoding types of the browser	gzip, deflate, br
accept_language	the set of natural languages that are preferred as response to the request	en-US,en;q=0.5
adblock	detect software blocking advertisements	false
audio_context_properties	list of audio context properties	{"ac_sampleRate":44100,"ac_state":"suspended","ac_maxChannelCount":0,"ac_numberOfInputs":1,"ac_numbe More >>
audio_input_devices	list of audio input devices	N/A
audio_output_devices	list of audio output devices	N/A
available_screen_resolution	the available screen resolution of the current monitor used	[2560,1414]
battery_info	information about the device battery	N/A
browser_build_number	build number of the browser for opera (buildNumber)	N/A
browser_buildID	build identifier of the browser (buildID)	20180316021607
browser_code_name	code name of the browser (appCodeName)	Mozilla
browser_engine	engine of the browser (product)	Gecko
browser_engine_build_number	build number of the browser engine (productSub)	20100101
browser_mime_types	acceptable MIME types	N/A
browser_minor_version	minor version of the browser (appMinorVersion)	N/A
browser_name	name of the browser (appName)	Netscape

Figure 1: Screenshot of ashara.cs.helsinki.fi

In addition to the challenges caused by some browsers API, the diverse nature of the browser ecosystem have been a big challenge as well. For instance, when a notification permission query is requested, Safari returns the response through a callback, while Firefox and Google Chrome return it as a promise [63, 64]. We consider each such case separately. This enabled us to not only target different versions and platforms and but it also helped us keep the front end code clean for easy troubleshooting and debugging.

4.2 Back End of Ashara

Ashara's backend consists of a data store and a Web server. The Web server accepts and serves the requests from the front end, and saves the collected data in the data store.

We implemented the Web server using the Node.js express framework [65]. The server is also responsible for extracting information from HTTP headers if HTTP headers are present in the client request. This information includes `accept-documents`, `accept-encoding`, and the client's IP address. We add these features to the features collected by the front end. The server also sets cookies (after asking the appropriate permissions) on the client's browser. This cookie contains only a unique session ID, which is used to track returning visitors. The session ID is computed using crypto module of Node.js. We used SHA-256 hash function and HEX encoding to generate the 256-bit hash string.

The data exchanged between the Web server and the front end is secured using HTTPS. We use Let's Encrypt certificates [66] for this purpose. Specifically, our server was hosted on "ashara.cs.helsinki.fi" domain, and we requested certificates from Let's Encrypt for this domain. Our Web server was deployed using a nginx reverse proxy [67].

The data collected by the front end and the back end are stored using the JSON format in a text file. We chose a text file as the format of choice for ease of development and analysis. The JavaScript Object Notation (JSON) format [68] is a text-based, language-independent data interchange format which is supported in a large number of programming languages such as Python and JavaScript.

5 Time Evolution of Browser Fingerprints

In this section, we have examined how the browser fingerprinting is evolving as the advancement of browser technology. New browser technologies, such as webRTC, WebAuido, and sensors, pave ways for new tracking mechanisms. Therefore, in order to examine this scenario, we tested a number of mobile browsers from Firefox and Google Chrome. In the following subsections, we have described the dataset and the measurement setup, and then experiment's results have presented.

5.1 Data Collection Methodology and Dataset Description

In this experiment, the dataset is gathered from Firefox and Google Chrome mobile browser's APKs. The two main reasons that we have chosen these browsers are: first, they are the most popular and used, according to StatCounterciteweb:BrowserMarket, Google chrome is the most popular browser worldwide with 56.27% and Firefox is 4th with 5.65%. Second, their APKs history files are readily available for downloads. Firefox keeps apk releases on ftp servers, `https://releases.mozilla.org/pub/mobile/releases/`, therefore, we use a bash script to automate downloading of the files. Whereas, in case of Google chrome, APKs are kept on a repository website, `https://www.apkmirror.com/apk/google-inc/chrome/`, and the website requires a user to download the files one by one, so we had to download the files manually. Table 2 summarizes our dataset.

Table 2: Dataset metadata

	Google Chrome	**Firefox**
First APK release date	2017-10-19	2017-09-28
Last APK release date	2015-04-15	2013-09-17
Total Number of APKs	20	32
Number of APKs tested	20	32
APKs version range	42-62	24-56

Each release of the APKs includes several variants in terms of architecture, language and version. For example, for a given Google Chrome release, one can find a number of compiled codes for different processor architecture, such as `x86`, `arm` and `arm64 + arm`. Similarly, this release may contain several languages and locale as well. However, in our dataset, we filtered out only arm, en-us, and non-beta latest version of the APKs from each release for both Firefox and Chrome.

These filtering variables are selected based on the facts: ARM architecture are largely supported on mobile phones; en-us is just language of the user interface and its effect on dataset is not significant; and non-beta latest versions are stable version that any change/fix in beta or earlier sub-versions are applied.

5.2 Measurement Setup

The measurement setup to extract information from APKs consists three main components; namely, two smart mobile phones, a development machine and a web server. First, the two smart phones are `Nokia 5` and `Xiaomi Redmi 1s`. The `Nokia 5` phone runs `Android 7` and `Xiaomi` runs `Android 4.4`; `Android 7` supports until API level 25 and Android 4.4 API supports until API level 19. Since API level support of the devices varies, we are not able to test all APKs in our dataset on single device. Therefore, older version of APKs tested on `Xiaomi Redmi 1s` and latest versions on `Nokia 5`. Second, the development machine can be any machine that runs Android debug bridge (ADB) command line utility to send commands to the mobile devices via USB. The third component is the web server which hosts the web browser fingerprinting website. The architectural topology of these components is shown in Figure 2. Basically, the figure have depicted that the development machine phone is connected to the mobile phone via USB and then the phone is connected to the web server over WiFi.

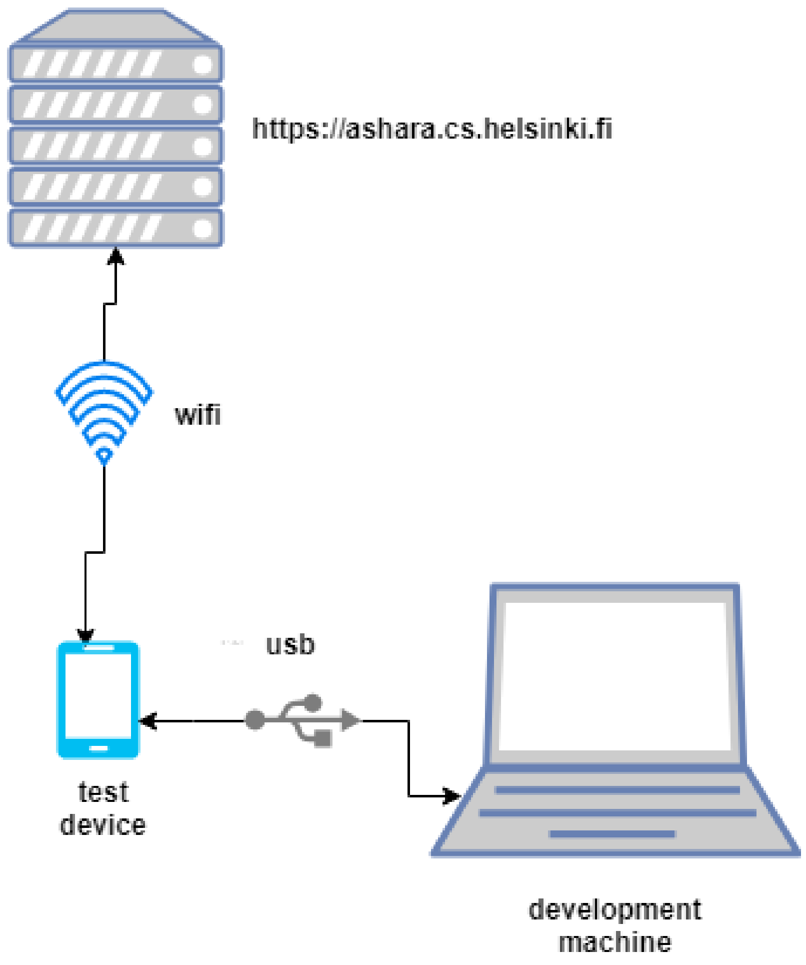


Figure 2: Measurement setup topology.

After the physically setup measurement components connections, and enable USB debugging on the devices, the testing process consisted of the following steps.

- Query for list of attached devices with development machine using the command `adb devices`.
- Install the apk using the command `adb install`.
- Start the web browser using the command `adb shell am start -a android.intent.action.VIEW -d https://ashara.cs.helsinki.fi`
- Uninstall the apk using the command `adb uninstall`.

In addition, since some fingerprinting algorithms include features that require the user to give permissions, we have given manually all permissions that are required during the test.

5.3 Results

5.3.1 Evolution of Permissions

Like other Apps, browser Apps require permissions to access user data and system features, such as contacts, camera, and the Internet. It is known that as technology advances, the permissions requirements of the browser Apps changes. In this subsection, we have examined how these changes in permission requirement evolve over time and how it affects user privacy, particularly for the Android platform. First, we have seen how we have acquired permission's information from APKs. Second, we have seen the growth of permission requirements over time for Google Chrome and Firefox browsers. Finally, we have examined the effect of minimum SDK and target SDK metadata on permission requirements of the Apps.

In an Android App, permission requirements are specified in a configuration file called `manifest.xml`. The manifest file includes <*uses-permission*> tags to specify the required permissions. Since Android applications are archived as an .apk file for deployment, Android SDK provides Android Asset Packaging Tool `(aapt)` to view internals of the package [69]. The `appt` tool includes a number of commands and options, such as `aapt dump badging` and `aapt dump permissions`. In our project, `appt dump badging` is used to retrieve version names, SDK versions, target SDK versions and install locations and `aapt dump permissions` command is used to get a list of required permissions.

It is known that the number of permissions required by the browsers is increasing over time. In order to see the evolution of permission requirements, we have examined one and half year Google Chrome APK versions history and close to four years Firefox APK versions history. The result of the experiment has clearly proven that the permission requirements of the applications have grown. For example, Firefox browser, as shown in Figure 4, permission requirements steady increase as version number changes. Similarly, Google Chrome has shown the same trend as shown in Figure 3.

On Android platform, permissions requirement are divided as normal and dangerous permissions [70]. Normal permissions contain little risk to user's privacy, whereas dangerous permissions expose privacy-sensitive information. In our experiment, we have observed that both Firefox and Chrome browser require both normal and dangerous permissions. For example, as shown in Figures 4 and 3, about half of the permission required by the majority of the APKs are dangerous permissions. Moreover, we have observed that both the mobile and desktop browsers do not require permissions for some hardware features, such as accelerometer and proximity sensors. However, many studies have shown that device sensors are subjected to security and privacy risks. For example, Mehrnezhad et al. [71] have shown that a users PIN code can be identified using motion and orientation sensors at a success rate of more than 74%.

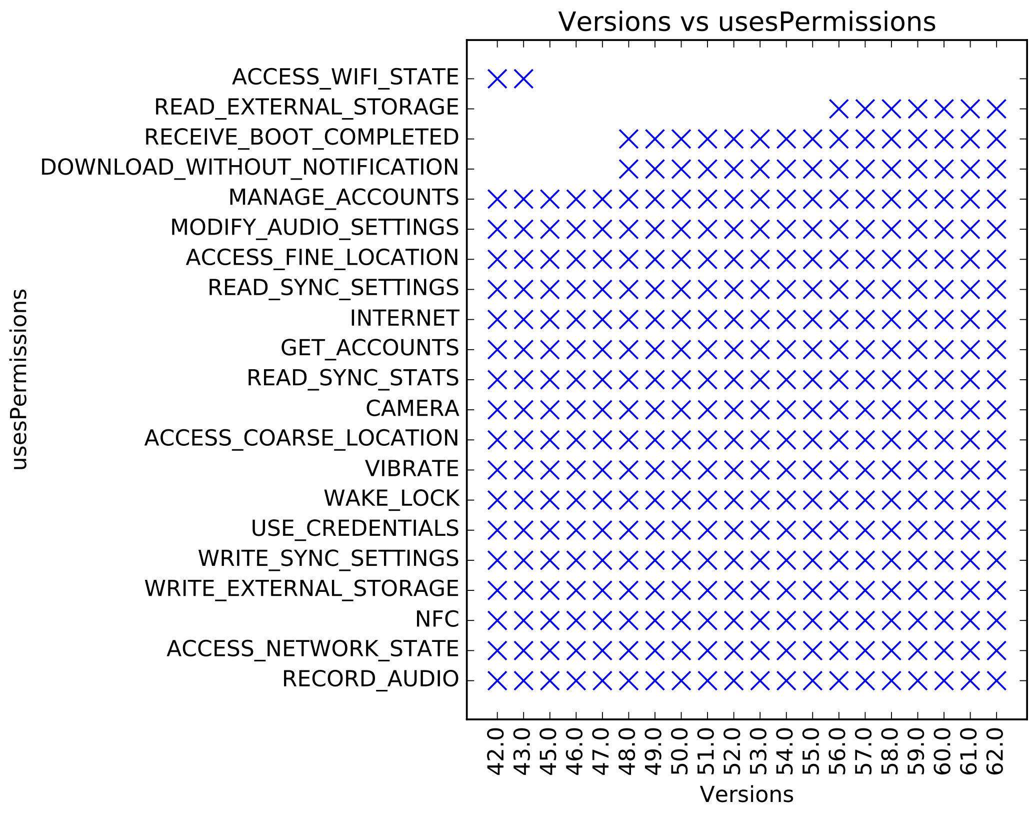


Figure 3: **Permissions used by different versions of Google Chrome.** The permissions requested by Google Chrome has been increased over time. For instance, version 42.0 released on 2015-04-15 requested 18 permissions while the latest version at the time of our analysis, version 62.0 released on 2017-10-19, requests 21 permissions.

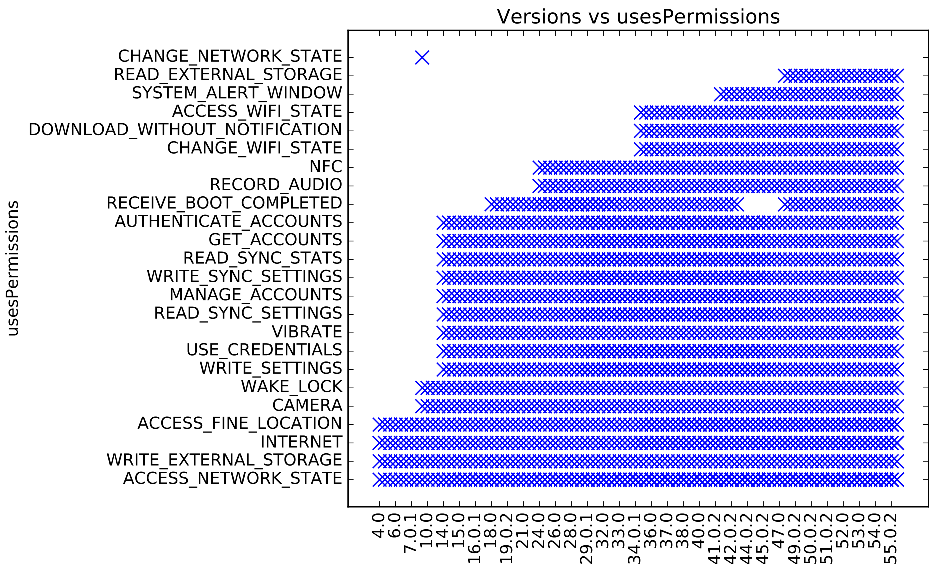


Figure 4: **Permissions used by different versions of Firefox.** The permissions requested by Firefox have been steadily increasing over time. For instance, version 4.0 released on 2013-09-17 requested only 4 permissions while the latest version at the time of our analysis, version 55.0 released on 2017-09-28, requests 24 permissions.

In addition to examining the permission requirement growth along versions, we have examined the privacy concerns related to the SDK level. As shown in the Figures 5 and 6, browsers tend to support the latest SDK versions. The latest versions are usually under heavy development process and the privacy concerns are left to be addressed later in the agile development process. Moreover, mobile platforms follow different approaches to deal with the permission requests of the applications. For example, Android OS have used install-time (Android 5.1.1 and below) and runtime permission (Android 6.0 and higher) request methods. Install-time permission request method asks the user to grant all the required permissions at install time. On the other hand, in the runtime permission request method, requests are issued at runtime when the user demands it. Run-time permission request method allows a user to fine grain control permissions. This method gives more leverage to the users to protect its privacy. However, the install-time method relays on the installed app implementations to control permission requests. Therefore, how to handle the privacy concerns is at the mercy of the app.

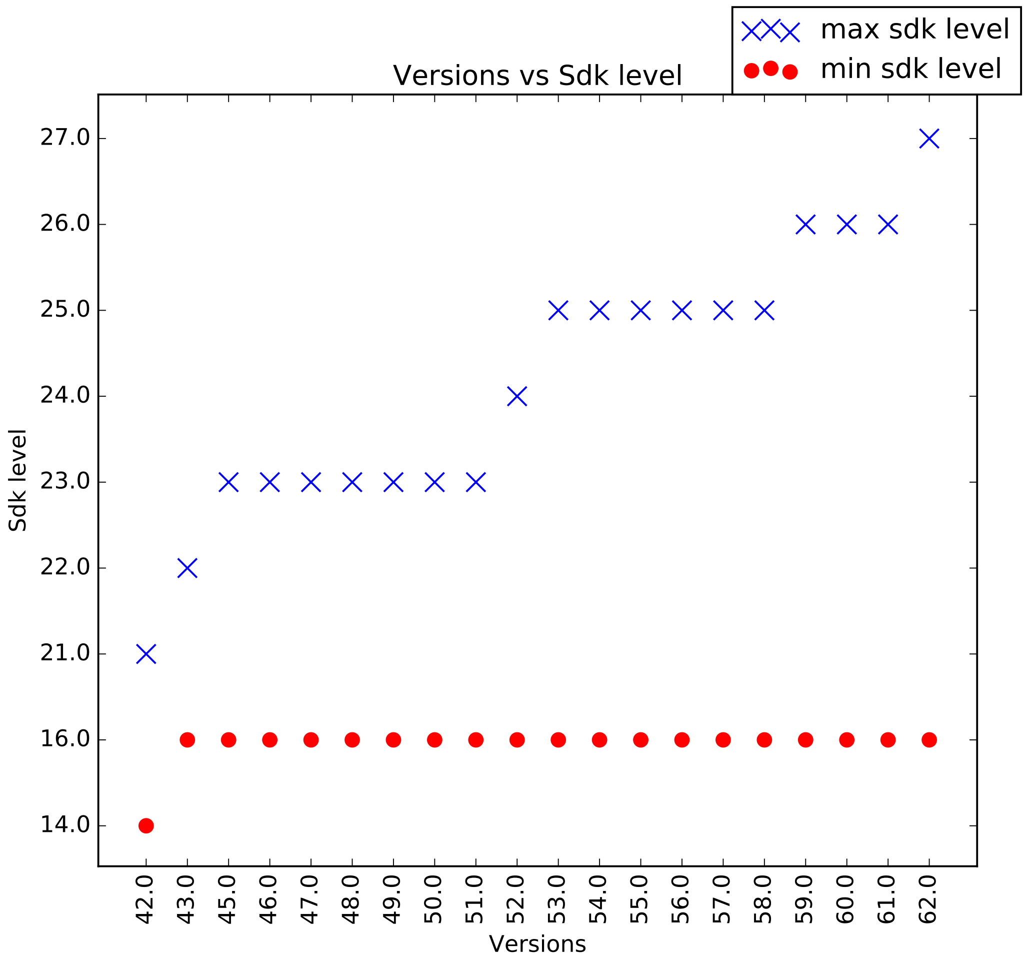


Figure 5: **Changes of minSDK and maxSDK with the version of Google Chrome.** For Google Chrome browser, maximum SDK level is higher than minimum SDK level for all versions in our dataset.

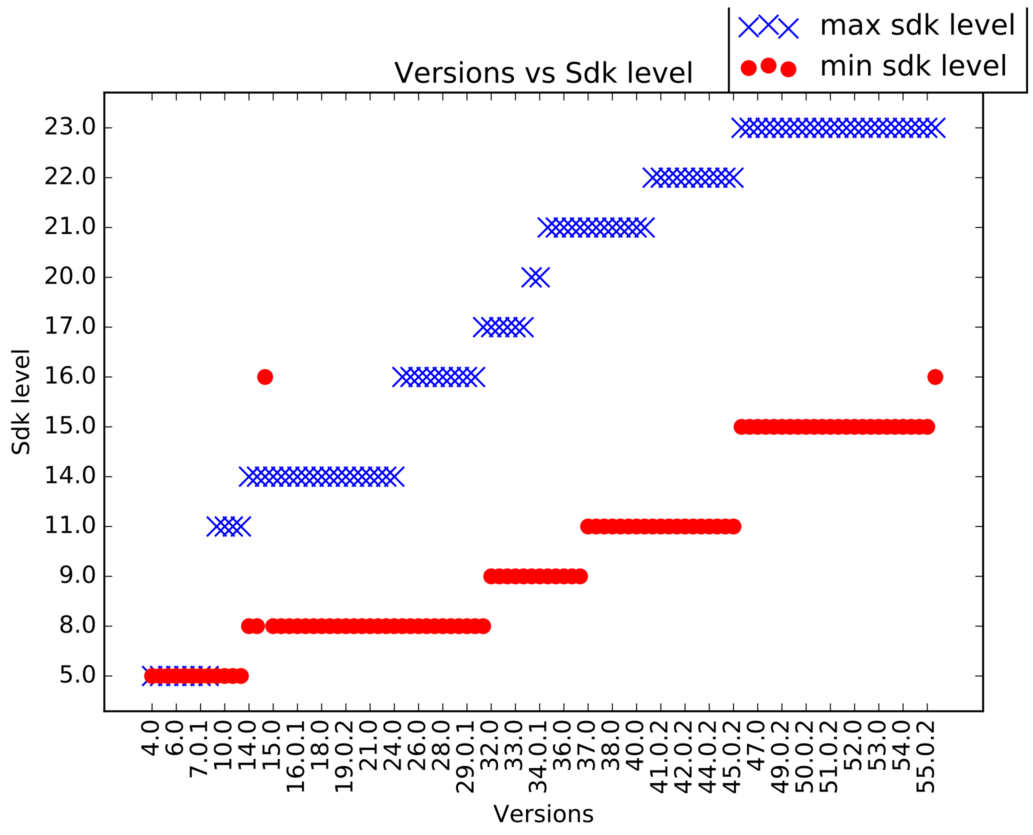


Figure 6: **SDK versions supported by the different versions of Firefox.** For Firefox browser, maximum SDK level is higher than minimum SDK level for all versions in our dataset after version 10.0.

5.3.2 Fingerprinting

In the previous subsection, we have briefly seen the effect of the granted permission requests and the SDK level in browser fingerprinting. In this subsection, we will present the result of the experiment. In this experiment, we have tested 20 Google Chrome and 32 Firefox browser versions. We have observed fingerprint changes of browser features over time. The experiment is done in control environment where we have used a fresh APK installs for each test. Thus, these results reflect default setting of the browsers. The result of the experiment has summarized in Table 3.

Feature	**Google Chrome**	**Firefox**	**Category**	**Fingerprinting Scope**
accept_documents	2	1	http headers	browser
accept_encoding	3	2	http headers	browser
accept_language	2	3	http headers	browser, user
ip_address_from_server	1	2	http headers	environment
available_screen_resolution	1	2	window object and navigator interface	device
battery_info	10	15	window object and navigator interface	environment
browser_build_number	1	1	window object and navigator interface	browser
browser_buildID	1	32	window object and navigator interface	browser
browser_code_name	1	1	window object and navigator interface	browser
browser_engine	1	1	window object and navigator interfaces	browser
browser_engine_build_number	1	1	window object and navigator interface	browser
browser_mime_types	1	1	window object and navigator interface	browser
browser_minor_version	1	1	window object and navigator interface	browser
browser_name	1	1	window object and navigator interface	browser
browser_vendor	1	1	window object and navigator interface	browser
browser_vendor_version	1	1	window object and navigator interface	browser
browser_version_and_platform	20	2	window object and navigator interface	device, browser
browser_version_number	1	1	window object and navigator interface	browser
color_depth	1	2	window object and navigator interface	device
cookies_enabled	1	1	window object and navigator interface	user
cpu_class	1	1	window object and navigator interface	device
do_not_track	1	1	window object and navigator interface	user
hardware_concurrency	1	2	window object and navigator interface	device
indexed_db	1	1	window object and navigator interface	browser
open_database	1	1	window object and navigator interface	browser
session_storage	1	1	window object and navigator interface	browser
local_storage	1	1	window object and navigator interface	browser
navigator_platform	1	2	window object and navigator interface	device, browser
network_information	2	3	window object and navigator interface	device, environment
operating_system_language	1	1	window object and navigator interface	user
os_and_cpu	1	2	window object and navigator interface	device
permissions_after_fingerprint	3	5	window object and navigator interface	user
permissions_before_fingerprint	2	3	window object and navigator interface	user
pixel_ratio	1	2	window object and navigator interface	device
preferred_language	1	1	window object and navigator interface	user
regular_plugins	1	1	window object and navigator interface	browser
screen_resolution	1	2	window object and navigator interface	device
touch_support	1	1	window object and navigator interface	device
user_agent	20	32	window object and navigator interface	device, browser
media_devices	17	18	media devices and WebRTC	device
audio_input_devices	20	18	media devices and WebRTC	device
audio_output_devices	1	1	media devices and WebRTC	device
video_input_devices	20	18	media devices and WebRTC	device
media_recorder_audio_mime_types	2	2	media devices and WebRTC	browser
media_recorder_video_mime_types	3	2	media devices and WebRTC	browser
has_microphone	1	3	media devices and WebRTC	device
has_speakers	2	2	media devices and WebRTC	device
has_web_cam	1	3	media devices and WebRTC	device
ip_address	1	3	media devices and WebRTC	environment
audio_context_properties	4	6	web audio context	device, browser
web_audio_fingerprint_oscillator	5	6	web audio context	device, browser
audio_oscillator_and_dynamicsCompressor	7	6	web audio context	device, browser
device_current_position	1	22	sensors	environment
device_light	1	32	sensors	environment
device_motion	13	32	sensors	environment
device_orientation	12	24	sensors	environment
device_proximity	1	1	sensors	environment
has_lied_browser	1	1	browser tampering	user
has_lied_languages	1	1	browser tampering	user
has_lied_os	1	1	browser tampering	user
has_lied_resolution	1	1	browser tampering	user
has_lied_resolution	1	1	browser tampering	user
os_name	1	2	miscellaneous fingerprinting techniques	device, browser
os_version	1	3	miscellaneous fingerprinting techniques	device, browser
canvas_fonts	1	1	miscellaneous fingerprinting techniques	device, browser
js_css_fonts	1	1	miscellaneous fingerprinting techniques	device, browser
is_private_browsing	1	2	miscellaneous fingerprinting techniques	user
adblock	1	1	miscellaneous fingerprinting techniques	user
canvas	9	14	canvas and webgl	device, browser
webgl	6	11	canvas and webgl	device, browser
webgl_renderer	1	2	canvas and webgl	device, browser
webgl_vendor	1	2	canvas and webgl	device, browser

Table 3: **List of features used for fingerprinting.** *For instance, `accept_documents` is in HTTP header category and it gives 2 and 1 unique fingerprint for Google Chrome and Firefox respectively. The scope of its fingerprinting is the browser itself.*

Basically, this table illustrates that how many unique fingerprints appear in the experiment for each feature and the specific contexts or environment where these fingerprints are most effective; namely, device, browser, user and operating system. The results of these experiments by category are presented as follow:

HTTP headers: In this category, the source of the fingerprint is HTTP headers. As shown in Table 3, the number of unique fingerprints for each feature is very low. For example, we have got only 2 and 3 unique `accept_language` fingerprints for Chrome and Firefox respectively. In general, the results show that as the browser's version changes, the HTTP header fields values remain the same. Thus, this experiment demonstrates that HTTP header fields do not reveal significant unique identifying information for fingerprinting, but they can be used with other fingerprint sources to create a user profile. On the other hand, the persistence of these fingerprints may help to establish a link between the new browser version fingerprint and previous ones, for instance, identifying updates.

In addition to examining how much identifying information one may obtain, we have explored the environment where the fingerprints are most effective. In HTTP headers category, we have observed that one may use these header fields to fingerprint browsers, users and the environment. First, accept document and encoding field could be used in fingerprinting the browser because these fields show the capacity of the browser to understand a different kind of documents or encoding. For example, in our dataset, we have seen that `sdch` encoding type is supported only on the Google Chrome browser. Second, accept languages field can be used in fingerprinting users and browsers. The values of this field reflect the user language preference. These preferences originate from both the user choice and activity. For example, one may guess that Google Chrome learns language preferences from several sources, such as language translates, visited websites languages, and location of the user. Thus, this value may be used in user fingerprinting. On the other hand, browser vendors, based on [14] standard, have implemented different algorithms to calculate the q value. These subtle differences open opportunity for browser fingerprinting. Finally, one may use the IP address header field to profile a user location.

Window object and Navigator interface: In this category, the source of identifying information is window object and navigator interfaces of the browser. Most of the features in this group exhibit no change as versions change. However, the user agent(Chrome and Firefox) and build ID(Firefox) strings are unique for each version. These strings include version specific identifying information, thus they are a rich source of entropy for a browser fingerprinting. In addition, for a given device, these strings differ in small substring between consecutive versions. For example, the user agent strings version 62 and 61 of Android 4.4.4 device differ by only small substring as shown in the bold text below:

Mozilla/5.0 (Linux; Android 4.4.4; HM 1S Build/KTU84P) AppleWebKit/537.36 (KHTML, like Gecko) ***Chrome/62.0.3202.73*** *Mobile Safari/537.36 Mozilla/5.0 (Linux; Android 4.4.4; HM 1S Build/KTU84P) AppleWebKit/537.36 (KHTML, like Gecko)* ***Chrome/61.0.3163.98*** *Mobile Safari/537.36*

Therefore, new browser fingerprint of a new version can be easily linked to the previous one by looking at the user agent. Similar to user agent string, build ID for Firefox browser may be used for this purpose as well. For example, `Android 7.1.2` version 56 and 55 `buildID` are *20170922214822* and *20170815231002* respectively and the difference between these two strings is in last 10 digits which refers to the exact build time.

Window and navigator objects of the browser could be used in fingerprinting the device, browser, user, and environment. As shown in the Table 3, there are several features that may be used in device fingerprinting, such as `screen_resolution`, `available_screen_resolution`, `cpu_class`, `color_depth`, `hardware_concurrency`, `pixel_ratio`, `os_and_cpu`, `touch_support`, `user_agent`, `navigator_platform`, `network_information`, `browser_version_and_platform`, and `pixel_ratio`. These entries give identifying information directly related to the underlying device. For example, screen resolution indicates that the potential area of the output device that can be used for a user interface rendering. For browser fingerprinting, We have `browser_build_number`, `browser_buildID`, `browser_code_name`, `browser_engine`, `browser_engine_build_number`, `browser_mime_types`, `browser_minor_version`, `browser_name`, `browser_vendor`, `browser_vendor_version`, `browser_version_and-_platform`, `browser_version_number`, `indexed_db`, `open_database`, `session_sto-rage`, `local_storage`, `navigator_platform`, `regular_plugins`, and `user_agent`. The identifying information that one may get from these entries is browser specific. For example, `regular_plugins` field lists installed plugins by default. In this category, for user fingerprinting, we have `cookies_enabled, do_not_track, permissions_after_fingerprint, permissions_before_fingerprint`, and `pre-ferred_language`. These fields profile the user actions. For example `permissions-_before_fingerprint` field holds a list of permissions granted by the user to a specific website before fingerprinting started. Similarly, we have several fields to fingerprint the environment the device running. These fields include `battery_info` and `network_information`. For example, `battery_info` gives information about the charge level while the user using the device.

Media devices and WebRTC: In this group, the source of the fingerprints is media devices API and WebRTC. These APIs give information about the installed audio/video devices on the system, such as the microphone, camera, and speaker and for each device, the media device information object includes prosperities, such as device id, group id, device kind and device label. For example, according to [52], device id is an HMAC of `device_id_salt`, origin, and raw id of the physical device. In our experiment, for a given device, we have observed that media devices ids are unique for each version due to the fact that each version of browser contains unique `device_id_salt`. Here, we have only examined characteristics of device id across browser versions and we will cover the behavior of device id across browsing session in our second experiment. Therefore, this uniqueness of device's id may help browser fingerprinting mechanisms to create a unique profile for users. Note that, as shown in the in Table 3, the number of unique fingerprints is less than the total number of the test because old versions of browser do not support media devices API or WebRTC. That is the reason, for `media_device` entry, we have got only 18 unique fingerprints from 32 Firefox browser versions tests.

As shown in Table 3, there are several fields that could be used in device fingerprinting, such as `media_devices`, `audio_input_devices`, `audio_output_devices`, `video_input_devices`, `has_microphone`, `has_speakers`, and `has_web_cam`. For example, `has_web_cam` field indicates the existence of a camera on the device. `Media_recorder_audio_mime_types` and `media_recorder_video_mime_types` fields give information about the mime types supported on the browser for media recording, thus this could be used in browser fingerprinting. The `ip_address` field may be used to learn the environment where the user is using a device, e.g. location.

Web audio context: First, we have observed that collection of audio properties values remain more or less the same across browser's versions. In our experiment, we have got 4 and 6 unique fingerprints for Chrome and Firefox respectively. These numbers do not indicate that values of collections properties are not same; For example, for chrome version 42 and 43, `audio_context_properties` entry differs only in orders as shown below:

Chrome 43: *'an_numberOfOutputs': 1, 'ac_numberOfOutputs': 0, 'ac_state': 'running', 'ac_channelInterpretation': 'speakers', 'an_fftSize': 2048, 'an_channelCount': 2, 'an_channelInterpretation': 'speakers', 'an_numberOfInputs': 1, 'ac_channelCount': 2, 'ac_maxChannelCount': 2, 'an_frequencyBinCount': 1024, 'an_channelCountMode': 'max', 'ac_channelCountMode': 'explicit', 'an_smoothingTimeConstant': 0.8, 'an_maxDecibels': -30, 'ac_sampleRate': 48000, 'an_minDecibels': -100, 'ac_numberOfInputs': 1*

Chrome 42: *'an_channelCountMode': 'max', 'ac_numberOfOutputs': 0, 'ac_state': 'running', 'an_minDecibels': -100, 'ac_channelInterpretation': 'speakers', 'an_fftSize': 2048, 'an_channelCount': 2, 'an_channelInterpretation': 'speakers', 'an_numberOfInputs': 1, 'ac_channelCount': 2, 'ac_maxChannelCount': 2, 'an_frequencyBinCount': 1024, 'ac_channelCountMode': 'explicit', 'an_maxDecibels': -30, 'ac_sampleRate': 48000, 'an_smoothingTimeConstant': 0.8, 'an_numberOfOutputs': 1, 'ac_numberOfInputs': 1*

In fact, there is also a few property's value changes across version, such as channelCount property is 2 from Chrome 42 to 59 but it is 1 from Chrome 60 to 62. Therefore, since these properties indicate the audio stack capacity of the host hardware, this can be used for hardware fingerprinting across versions.

Second, we have analyzed the frequency response of the audio stack of the browser by applying triangle waves that are generated by an oscillator node and an oscillator node with a dynamic compressor. For both entries, less than 50% of tests have resulted in unique fingerprints and we have observed that these fingerprints are not consistent across the whole dataset. whereas, we have seen some range of consecutive versions have the same values for both Firefox and Chrome as shown in Table 4 and

5; For example, in Firefox browsers dataset, from version 56 to 45, 44 to 42, 40-37, and 36 to 25 have the same fingerprints. We do not have enough volume of data to conclude that these fingerprints are unique for each version or consistency across versions in this experiment.

In this category, since values of these fields depend on both the underlying hardware audio stack and the browser software, these entries could be used to fingerprint both browser and device. For example, oscillator node fingerprinting.

Versions	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	43	42
Audio_context_properties	🟨	🟨	🟨	🟥	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟪
Audio_oscillator	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟥	🟥	🟩	🟥	🟥	🟨	🟥	⬛	⬛	⬛	⬛	🟦
Audio_oscillator_and_dynamicsCompressor	🟨	🟨	🟨	🟥	🟦	🟨	🟥	🟨	🟥	🟩	🟥	🟩	⬛	⬛	🟫	⬛	🟪	🟥	🟥	🟥
Canvas	🟨	🟨	🟨	🟩	🟥	🟥	🟥	🟫	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟩	🟩	🟪	🟦	⬜
Webgl	🟨	🟨	🟨	🟨	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟫	🟩	🟩	🟦	🟥	🟥

Table 4: **Consistency of fingerprints across versions of Google Chrome.** *For instance, the same canvas fingerprint was observed from version 48 to version 54. Similarly, the same WebGL fingerprint was observed from version 49 to version 58.*

Versions	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	26	25	24
Audio_context_properties	🟨	🟨	🟨	🟨	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟦	🟦	🟦	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	⬛
Audio_oscillator	🟨	🟨	🟨	🟨	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	⬛
Audio_oscillator_and_dynamicsCompressor	🟨	🟨	🟨	🟨	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟦	🟦	🟦	🟦	🟦	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	⬛
Canvas	🟨	🟨	🟥	🟥	🟩	🟦	🟪	🟪	🟪	🟥	🟩	🟥	🟥	🟥	🟥	🟧	🟥	🟥	🟥	🟥	🟦	🟦	⬛	⬛	⬛	⬛	⬛	⬛	⬛	⬜	🟧	🟧
Webgl	🟨	🟥	🟥	🟥	🟩	🟩	🟦	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟥	🟥	🟩	🟥	🟧	🟧	🟦	⬛	⬛	⬛	⬛	⬛	⬛	⬛

Table 5: **Consistency of fingerprints across versions of Firefox.** *For instance, the same canvas fingerprint was observed from version 34 to version 28. Similarly, the same WebGL fingerprint was observed from version 49 to version 39.*

Sensors: Data from the sensors is rather volatile, the luminosity or the position of the device can change drastically within seconds and is not persistent. There is a way to exploit this kind of information for fingerprinting purposes by using machine learning to build, e.g. the profiles of user movements. Therefore, sensors could be used to fingerprint the browser environment. But in this study, we don't have sufficient volume of data for this approach.

Browser tampering: In this experiment, we have not changed the default settings, so we have got consistent results across versions. We have not got discriminatory information for fingerprinting purposes. However, this kind of information may not provide a lot of data for fingerprinting for the majority of users, but those few who change the default settings/fake user agent will be very distinct. Thus, these fields could be used for user fingerprinting.

Miscellaneous fingerprinting techniques: In this category, we have employed several fingerprinting techniques to garner identifying informations, such as operating system name and version, list of installed fonts, mode of browsing and the existence of AdBlocker. In this experiment, since we have applied these techniques on newly installed browsers for each versions, we have got consistence results. Even if these results do not give us discriminatory information about the browse, we could use these results as benchmark for our second experiment which is done on data gathered from users.

In this category, we have seen several techniques to fingerprint device, browser, and user. `Is_private_browsing` and `adblock` can be used to reveal the user actions, so they are user fingerprinting. On the other hand, `os_name, os_version, canvas_fonts`, and `js_css_fonts` fields give identifying information that is related to browser and device. Therefore, these fields may be used in browser and device fingerprinting.

Canvas and WebGL: Canvas and WebGL fingerprints depend both on hardware and software version. In this category, we have tested canvas and WebGL fingerprinting techniques, and methods that reveal WebGL renderer and version information from WebGL debug facility. In this experiment, we have observed that canvas and WebGL fingerprinting techniques shows a similar trend as audio context fingerprinting. They are not consistent across the entire data set, whereas we have seen range of consecutive versions have the same fingerprint values as shown in Table 4 and 5. As an example, in chrome browsers, from version 62 to 59 have the same fingerprint. In addition, as shown in Table 3, the WebGL render and version information is consistent across versions. However, these features do not reveal a significant amount of information about the device, but it may be used in combination with other features, such as canvas and WebGL, to create unique fingerprints.

Similar to Web audio context category, all entries of this group's value depend on both the underlying hardware and the browser software, so these entries could be used to fingerprint both browser and device. For example, canvas fingerprinting.

6 Fingerprinting in the Wild

In this section, we have examined fingerprinting in the wild. Thus, we have built a fingerprinting tool, as it is discussed in Section 4, to collect data from the user's browsers and we have calculated the "self-information" or "entropy" of each feature and the expected value of the entropy of the features over all browsers. In the following subsections, first, we have described our dataset; second, we have presented the result of the mathematical analysis of the data.

6.1 Dataset Description

In this experiment, we have gathered a small set of browser's fingerprint data from users who visited our test website, `https://ashara.cs.helsinki.fi`. The total number of user's fingerprints is 134 and 96 are unique. In order to identify unique fingerprints, we have set cookies with unique session ID on the user browsers and so any browser subsequent visits that have the same session ID is considered non-unique. In the unique dataset, mobile phone browsers fingerprints account for 51 fingerprints. In addition to platform difference, we have seen geographical and demographical data bias in our dataset as well. Geographically, most of our fingerprint data have come from local sources, Nordic countries and demographically, the dataset is biased toward university students.

6.2 Results

In this subsection, we have presented the mathematical treatment of the dataset and the result of the analysis. In mathematical treatment, we have implemented the self-information (entropy) and the expected value of entropy formulas to show how much identifying information a given browser features contribute in our browser fingerprint algorithm. According to 'How unique is your web browser' research paper [3], the entropy of a particular output from an algorithm is given by

$$I(F(x) = f_n) = -log_2(P(f_n)). \tag{1}$$

This entropy information indicates the amount of identifying information that can be gathered form features of a browser (in bits). In addition to entropy, we have calculated the expected value of the entropy over our dataset and it is given by

$$H(F) = -\sum_{n=0}^{N} P(f_n)log_2(P(f_n)). \tag{2}$$

In the data analysis, we have examined the entropy of each features individually and in conjunction with each other. First, we have calculated the entropy of a browser features and the expected value of the entropy over all browsers and the results are summarized in Table 6 and 7. Second, we have taken concatenation of pair of features fingerprints in our dataset distribution and we have calculated the expected entropy and normalized expected entropy and the result is summarized in Table 7 and 8.

6.2.1 Entropy of Browser's Features per User

In Table 6, we have calculated the entropy information of a user's browser[1]. As show in the table, we have 72 features and the corresponding entropy information. Based on the entropy information, we can group these features into three levels; namely, high, medium, and low. All features that give six and more than six bits of information are included in identifying information rich high level. The medium level contains features that offer between six and three bits of information and the low level includes all the remaining features that give three and less than three bits of information.

In the high level group, we have features that can be divided into different subgroups as well. In the first subgroup, we have the volatile features such as `device_orientati-on, device_motion, device_current_position, battery_info` and `ip_address`. These features, even if they are rich in entropy information, we may not include them in creating stable fingerprint ID, because the fingerprints that are gathered from such sources are not reproducible. However, these features can be used in fingerprinting by implementing advanced machine learning techniques and ad-hoc methods. For example, according to Olejnik *et al.* [36], the battery status API information can be used in fingerprinting web users for short time period by observing the battery level of the device. The IP addresses may reveal significant amount of information about the context of the web user, such as location and the presence of proxies. In the second subgroup, we have `webgl`, `web_audio_fingerprint_oscillator, video_input_devices, audio_input_devices`, and `media_devices`. Unlike the volatile features, these features can be used to create a consistent fingerprint ID. For example, WebGL, according to [61], gives entropy of 4.30 bits, over 300 samples, but in our research, we have got 6.58 bits of information over 96 samples for this particular user. Similarly web-audio oscillator, video input devices, audio input devices and media devices are rich in entropy. Specially, these features, in addition, shows that the more new APIs are added to web ecosystem, the attack surface for fingerprinting increases significantly. In the third subgroup, we have `user_agent` and `accept_language`. The user agent contains information about the browser type, minor versions and others. In our research, we have proved that the use agent field is still highly discriminant feature. Surprisingly, we have found that the `accept_language` field of HTTP request is rich in entropy and we have got 6.58 bits of information over 96 samples for this particular user. As it is discussed in Section 3.1 accept language field shows the user language preferences and it is given as list of languages with quality factor. For example, the accept language string for this

[1]We have acquired user consent to use the data in our explanation.

particular user is: ‘en-GB,hi-IN;q=0.8,en-US;q=0.6’. In this string, ‘hi-IN;q=0.8’ sub-string is highly discriminating chunk. Since, for this case, we are aware of the location of the user from `timezone_offset` property, so we can easily profile the user using this sub-string, like this web user prefers Hindu next to British English. To sum up, one could say that these entropy information rich features can be used in browser fingerprinting independently or with other lower entropy level features.

In medium level group, we have several features, such as `canvas`, `timezone_offset`, `os_version`, `permissions_after_fingerprint`, `webgl_renderer`, `web_audio_fingerprint_oscillator_and_dynamicsCompressor`, and `audio_context_properties`. These features can be sub-grouped as volatile and non-volatile. The volatile subgroup includes `timezone_offset` and `permissions_after_fingerprint`. The timezone offset fingerprints are not stable. It can be change if the user move to new time zone or because of daytime saving time changes. Similarly, `permissions_after_fingerprint` is subjected to changes if the user decide to alter the existing permission state. As it is discussed in Subsection 3.2, the browsers gives range of permissions for websites with and without consent of the user. In our dataset, `permissions_after_fingerprints` field contains list of browser permissions status, e.g. `‘webcam’: ‘granted’, ‘microphone’: ‘granted’, ‘midi’: ‘granted’, ‘geolocation’: ‘granted’, ‘notifications’: ‘granted’, ‘push’: ‘granted’`. In non-volatile subgroup, we have `os_version`, `webgl_renderer`, `web_audio_fingerprint_oscillator_and_dynamicsCompressor`, and `audio_context_properties`. Basically, these features closely related to the hardware and software configuration of the host device, therefore, they can be used to profile a user consistently. In general, as shown in Table 6, this group of features contains less entropy information, thus they should be conjugated with other level of groups to offer enough entropy information in a fingerprinting algorithm.

In low level group, we have all the remaining features that are not included in high and medium level groups. This group includes features that are derived from several sources, such as HTTP requests, permission requests, sensor APIs, browser storage APIs, browser tampering, media device information requests and browser navigator and windows interfaces. For example, we have `accept_encoding` that gives 2.19, `has_speakers` that give 0.88, `cookies_enabled` that gives 0 bits of surprisal information and so on. This group of features exhibit low level of identifying information because of several reasons. First, there are features that most browser implementations give standardized and similar API outputs, such as `accept_documents` and `regular_plugins`. Second, in some features, users rarely change the default settings and the values are consistent across platforms, e.g. `is_private_browsing,` `has_lied_browser` and `indexed_db`. In general, this group of features entropy information level so low, therefore, one cannot use these features independently in fingerprinting.

Source	Entropy
webgl	6.58
web_audio_fingerprint_oscillator	6.58
video_input_devices	6.58
user_agent	6.58
media_devices	6.58
ip_address_from_server	6.58
ip_address	6.58
device_orientation	6.58
device_motion	6.58
device_current_position	6.58
browser_version_and_platform	6.58
battery_info	6.58
audio_input_devices	6.58
accept_language	6.58
web_audio_fingerprint_oscillator_and_dynamicsCompressor	5.0
permissions_after_fingerprint	5.0
webgl_renderer	4.58
timezone_offset	4.0
os_version	4.0
canvas	4.0
audio_context_properties	3.13
pixel_ratio	3.0
audio_output_devices	2.5
network_information	2.42
permissions_before_fingerprint	2.34
accept_encoding	2.19
navigator_platform	2.13
webgl_vendor	1.94
screen_resolution	1.58
preferred_language	1.58
available_screen_resolution	1.58
touch_support	1.5
js_css_fonts	1.46
canvas_fonts	1.46
hardware_concurrency	1.3
os_name	1.16
color_depth	1.16
accept_documents	1.0
has_speakers	0.88
media_recorder_video_mime_types	0.86
media_recorder_audio_mime_types	0.86
browser_vendor	0.78
browser_mime_types	0.78
regular_plugins	0.68
open_database	0.54
has_web_cam	0.5
browser_engine_build_number	0.42
os_and_cpu	0.38
browser_buildID	0.38
adblock	0.32
do_not_track	0.3
has_microphone	0.28
has_lied_languages	0.23
device_light	0.14
is_private_browsing	0.13
has_lied_os	0.13
has_lied_browser	0.13
indexed_db	0.09
has_lied_resolution	0.09
device_proximity	0.08
session_storage	0.06
local_storage	0.06
operating system_language	0.03
cpu_class	0.03
browser_minor_version	0.03
cookies_enabled	-0.0
browser_version_number	-0.0
browser_vendor_version	-0.0
browser_name	-0.0
browser_engine	-0.0
browser_code_name	-0.0
browser_build_number	-0.0

Table 6: **An example of entropy calculated for browser features of a single user.** *For instance, for this given user, the* `webgl` *feature gives 6.58 bits of identifying information.*

6.2.2 Expected Value of Feature's Entropy Over All Browsers

In addition to the entropy information of a single user, we have calculated the expected value of the entropy as shown in Table 6. The expected value of a feature, entropy, is the long-run average value of the feature's entropy over our dataset. According to our dataset, the maximum expected entropy gains should come from user_agent, 6.1 bits and followed by `ip_address_from_server`, 5.98, and media_devices, 5.97. Note that `ip_address_from_server` is the IP address of the client that is extracted from HTTP request and logged at server. In our experiment, 19 features have given more than 4.0 bits of entropy, 16 features have given between 4 and 2 bits identifying information, 30 features have given between two and zero, and seven features have given 0 bits of information. In the Table 7, we can clearly see that features that are derived from the modern APIs, such as media devices, web audio, and permission query, sensors and accept language field of the HTTP request, have given comparable amount of bits of information to that of canvas, webgl, user agent and IP address. In addition, we have seen that additional identifying information can be obtained by simply including permission requests in a web applications. For example, in our dataset, the expected entropy gain form a Web browser with default or existing permission setting is 2.57, but this value increases to 4.5 after the user have responded to permission requests. Furthermore, we have seen that cookies are enabled for all browser in our dataset, thus, even if modern browsers have enough cookie management tools, cookie based profiling is still viable alternatives. In general, from Table 7, we have understood that a website can create stable fingerprint ID if it is able to fingerprint the entropy rich and some other features.

Source	**Entropy**
user_agent	6.1
ip_address_from_server	5.98
media_devices	5.97
browser_version_and_platform	5.86
audio_input_devices	5.86
canvas	5.85
webgl	5.82
web_audio_fingerprint_oscillator_and_dynamicsCompressor	5.48
web_audio_fingerprint_oscillator	5.31
ip_address	5.27
webgl_renderer	5.25
video_input_devices	5.16
accept_language	4.76
permissions_after_fingerprint	4.5
audio_context_properties	4.48
available_screen_resolution	4.34
device_current_position	4.3
os_version	4.08
device_orientation	4.06
device_motion	3.91
js_css_fonts	3.85
screen_resolution	3.65
canvas_fonts	3.56
battery_info	3.49
webgl_vendor	3.21
navigator_platform	3.0
browser_mime_types	2.58
permissions_before_fingerprint	2.57
audio_output_devices	2.47
timezone_offset	2.41
network_information	2.38
pixel_ratio	2.24
regular_plugins	2.2
os_name	2.07
hardware_concurrency	2.04
touch_support	1.89
preferred_language	1.7
browser_buildID	1.68
accept_documents	1.65
accept_encoding	1.57
media_recorder_video_mime_types	1.44
media_recorder_audio_mime_types	1.44
color_depth	1.41
browser_vendor	1.41
os_and_cpu	1.38
has_speakers	0.99
browser_engine_build_number	0.91
open_database	0.9
adblock	0.9
has_web_cam	0.87
do_not_track	0.84
device_light	0.75
has_lied_languages	0.74
has_microphone	0.67
has_lied_os	0.48
has_lied_browser	0.48
is_private_browsing	0.41
device_proximity	0.37
indexed_db	0.34
has_lied_resolution	0.34
session_storage	0.25
local_storage	0.25
operating system_language	0.15
cpu_class	0.15
browser_minor_version	0.15
cookies_enabled	-0.0
browser_version_number	-0.0
browser_vendor_version	-0.0
browser_name	-0.0
browser_engine	-0.0
browser_code_name	-0.0
browser_build_number	-0.0

Table 7: **Expected value of feature's entropy in our dataset.** *For instance, for our dataset, the expected value of the entropy for the* `user_agent` *is 6.1 bits.*

6.2.3 Entropy of Joint Features

In the previous sub-subsection, we have seen the entropy information gain form each feature individually. In this sub-subsection, first, we will examine the entropy gain by pairing feature's fingerprints; Second, we have examined the entropy gain by concatenating all fingerprints of the features into single string.

As show in Table 7, we have calculated the paired features entropy gain and presented by 72 x 72 table. Basically, the table shows the entropy gain by pairing features. For example, `user_agent` and `ip_address_from_server` individually give 6.1 and 5.98 bits of information respectively, but when we pair them together, the entropy gain is increased to 6.5 bits. The average entropy information that we got from individual features is 2.37 bits and the maximum entropy gain is 6.1 bits, from `user_agent`. However, the average and maximum entropy gain for paired features has increased to 3.88 and 6.58 respectively. The maximum entropy is gained from `audio_input_devices` and `user_agent` pair. In fact, the maximum entropy that we can get from a dataset size of 96, if the all features have unique fingerprints is

$$-log_2(1/96)$$

which is 6.58 bits. Therefore, as shown in Table 7, we can get this amount of entropy information by just joining `audio_input_devices` and `user_agent` or `media_devices` and `user_agent` pairs; this means, if we take these pair of features, it is enough to identify all users in our dataset uniquely.

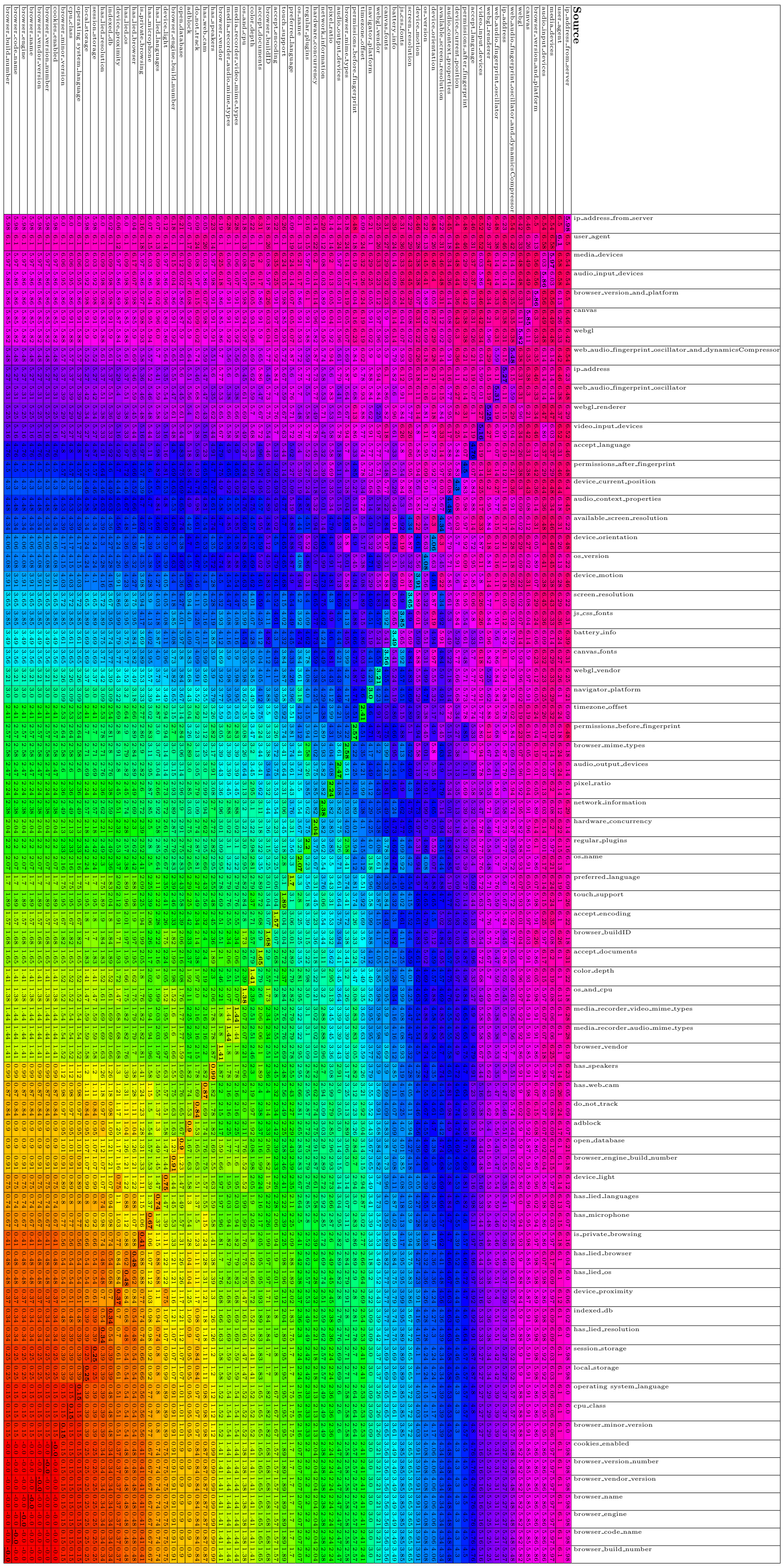


Figure 7: **Entropy of joined features.** *For instance, for our dataset, joint expected value of entropy for* `user_agent` *and* `media_devices` *is 6.58.*

In addition to the vanilla paired features entropy gain calculation, we have calculated normalized paired features entropy gain, as shown in Table 8, to see the change in entropy gain and we have found that the gain is low. For example, if we see the top three entropy rich features pairs (the first three rows in Table 8), we get maximum 10% increase in entropy gain. The main reason behind this low increase is: our dataset size is small, so the chance to get a repetitive fingerprint is low, like the same user agent string that is shared by many users, and most of the feature's fingerprints are unique by themselves. Therefore, joining two feature's fingerprints, which are unique by themselves at first place, does not increase their uniqueness. In spite of low entropy gain, there are several advantages in joining to fingerprints together. First, it increases the robustness of the fingerprint, for example, if two users have the same user agent, we can use IP addresses to differentiate them. Second, it can be used to re-identify (detect updates). As you can see in Table 4 and 5, some features show the same fingerprint across versions, thus, if a user update the browser, we can easily link new versions of the browser to old one just by using user agents.

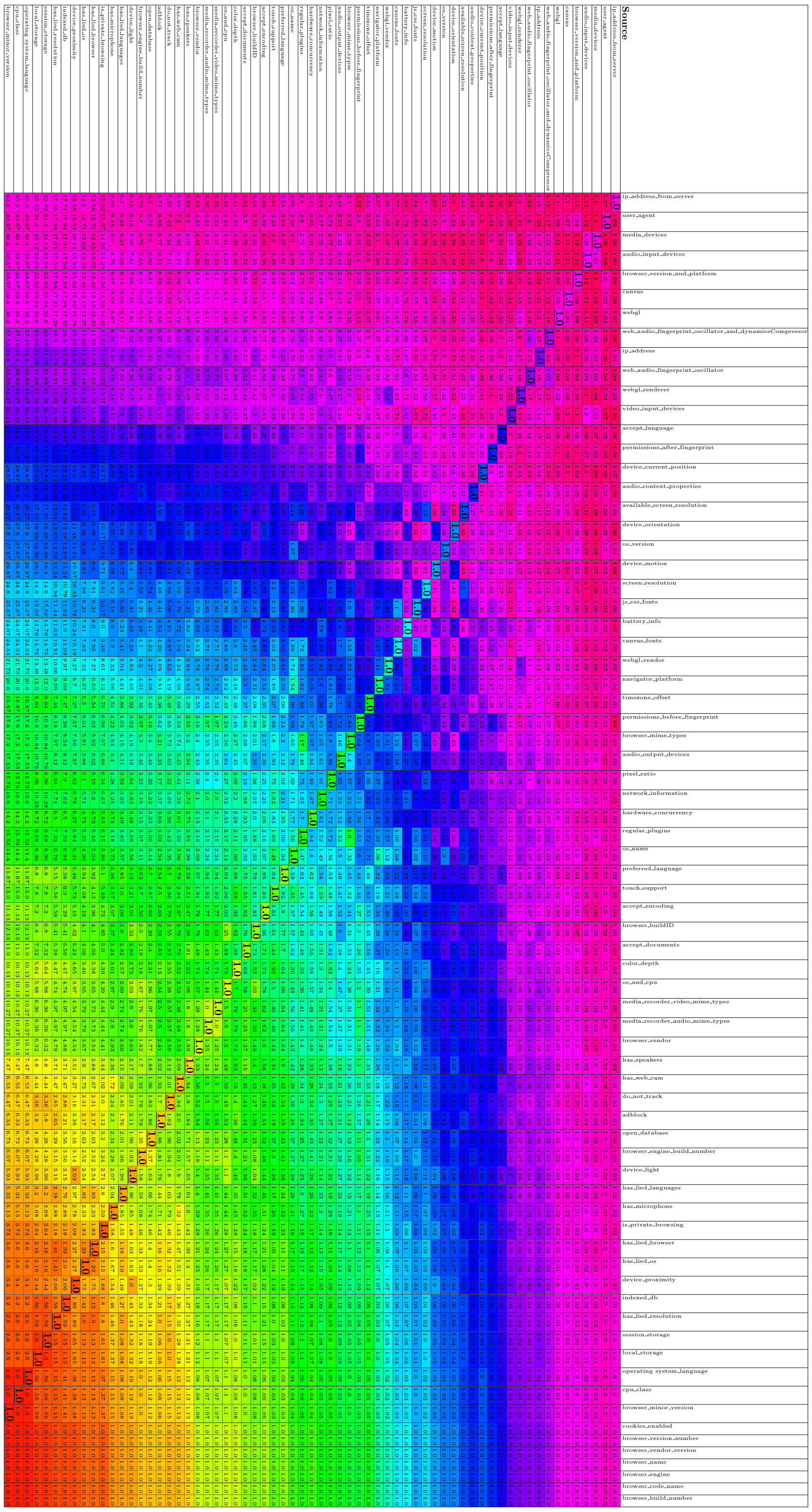


Figure 8: **Normalized joint entropy.** *For instance, for our dataset, expected value of entropy for* `user_agent` *has increased by 10% when it is joined with* `media_devices`*.*

Furthermore, we have examined the joint entropy gain for all feature's fingerprints together, without the volatile features, and without volatile and browser storage linked features in our dataset. When we join fingerprints of all features together, the average and maximum entropy gain is 6.58, thus all users are unique. For all joint fingerprints without the volatile features, such as IP addresses, sensor reading, battery info and permission queries, the average and maximum entropy gain is 6.58 as well. In the case of without volatile and browser storage linked joint fingerprints, we have removed all media related features, such as media media_devices, `audio_input_devices`, `video_input_devices`, and `audio_output_devices`, and the average entropy gain is decreased to 6.56. The number of unique fingerprint is decreased by one, 95 out of 96.

To sum up, we have observed that because of the low dataset size, the change in entropy gain from single features to all feature's joint is minimal. We have got a maximum of 6.1 bits from individual feature's fingerprints and 6.58 bits by joining all feature's fingerprints together. Therefore, our experiment shows that it is enough to take set of entropy rich features, such as user agent, IP addresses, and media devices, to identify a user uniquely rather than collecting fingerprints from several sources.

7 Conclusions

In this thesis project, we have gathered 72 fingerprintable features from different sources and we have used them to investigate time evolution of mobile browsers and fingerprinting in the wild. While studying time evolution of mobile browsers, we have investigated the growth of permission requirements of the browsers and the effect of SDK levels on feature's fingerprints. The permission requirements of the browsers have increased significantly over time for both Firefox and Google Chrome. We have seen that browsers tend to support the maximum SDK level that exists at the time of release and usually, privacy and security requirements are addressed in agile fashion.

In fingerprinting in the wild, we have gathered fingerprints from 134 browsing sessions of which 96 are unique. For our dataset size, we have observed that the maximum entropy gain (6.58 bits), can be obtained from joint of two high entropy features, e.g. media devices and use agent string. In general, the main achievements of this thesis project are the inclusion of modern browser APIs, such as sensors, audio, media devices and battery and the focus on mobile phones.

Contrary to the results, there are several shortcoming in this thesis project. First, the dataset size is rather small and it is very hard to draw conclusions based on it. Second, the data itself is not spread evenly to general population; mainly, the collected data is biased to university students. Third, geographically, the majority of the data is gathered from Finland. Finally, the study is not focused on a specific feature for deep understanding.

In this thesis project, we have understood that in the feature work, there are several areas that need to be explored. First, our dataset size is small, so more data should be gathered. Second, we have calculated entropy gain for only pairs of features, one can extend it to more than two features to see how the entropy gain changes. Third, we have collected the sensor data, but we have not studied deep enough how to use it in device fingerprinting. Finally, we have collected fingerprints from 72 features, but in future work, one may take the subset of this features and do a deeper study to find a better way to use them in fingerprinting mechanisms.

In conclusion, this work shows the increasing importance of the problem of fingerprintability of browsers when more information is becoming available to the websites through the browser APIs. Browser vendors are aiming to support the latest features, and provide access to all available resources of mobile devices without necessary bearing privacy in mind.

References

1 J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. Van Randwyk, and D. Sicker, "Passive data link layer 802.11 wireless device driver fingerprinting," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

2 T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," *IEEE Trans. Dependable Secur. Comput.*, vol. 2, pp. 93–108, Apr. 2005.

3 P. Eckersley, "How unique is your web browser?," in *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, (Berlin, Heidelberg), pp. 1–18, Springer-Verlag, 2010.

4 S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 1388–1401, ACM, 2016.

5 F. Alaca and P. C. van Oorschot, "Device fingerprinting for augmenting web authentication: Classification and analysis of methods," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, USA), pp. 289–301, ACM, 2016.

6 "Fingerprinting guidance." `https://www.w3.org/TR/fingerprinting-guidance/`. Accessed: May 3, 2018.

7 G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 674–689, ACM, 2014.

8 N. M. Al-Fannah and W. Li, "Not all browsers are created equal: Comparing web browser fingerprintability," *CoRR*, vol. abs/1703.05066, 2017.

9 H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, "Mobile device identification via sensor fingerprinting," *CoRR*, vol. abs/1408.1416, 2014.

10 "Browser tampering." `https://github.com/Valve/fingerprintjs2/wiki/Browser-tampering`. Accessed: January 9, 2018.

11 "DetectRTC." `https://github.com/muaz-khan/DetectRTC/blob/master/DetectRTC.js`. Accessed: January 11, 2018.

12 "AudioContext fingerprint test page." `https://audiofingerprint.openwpm.com/`. Accessed: January 31, 2018.

13 P. Laperdrix, W. Rudametkin, and B. Baudry, "Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints," in *37th IEEE Symposium on Security and Privacy (S&P 2016)*, (San Jose, United States), May 2016.

14 R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1," RFC 2616, RFC Editor, June 1999. `http://www.rfc-editor.org/rfc/rfc2616.txt`.

15 "HTTP-headers." `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers`. Accessed: December 12, 2017.

16 A. Petersson and M. Nilsson, "Forwarded HTTP extension," RFC 7239, RFC Editor, June 2014.

17 "X-forwarded-for." `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For`. Accessed: December 12, 2017.

18 European Computer Manufacturers Association , "Standard ECMA-262: ECMAScript language specification," vol. 2, p. 2002, 1999.

19 "HTML-standard." `https://html.spec.whatwg.org/multipage/system-state.html#the-navigator-object`. Accessed: December 16, 2017.

20 "Navigator." `https://developer.mozilla.org/fi/docs/Web/API/Navigator`. Accessed: December 17, 2017.

21 "User agent switcher." `https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher/`. Accessed: May 21, 2018.

22 "CSSOM view module." `https://drafts.csswg.org/cssom-view/#the-screen-interface`. Accessed: December 27, 2017.

23 "Time zone offset." `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/getTimezoneOffset`. Accessed: December 27, 2017.

24 A. Barth, "HTTP state management mechanism," RFC 6265, RFC Editor, April 2011. `http://www.rfc-editor.org/rfc/rfc6265.txt`.

25 "HTTP cookies." `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies`. Accessed: January 4, 2018.

26 "Session storage." https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage.

27 "Session storage." `https://html.spec.whatwg.org/multipage/webstorage.html#dom-sessionstorage`. Accessed: January 5, 2018.

28 “Fingerprinting guidance.” `https://w3c.github.io/fingerprinting-guidance/`. Accessed: January 7, 2018.

29 “Local storage.” `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage`. Accessed: January 5, 2018.

30 “Local storage.” `https://html.spec.whatwg.org/multipage/webstorage.html#dom-localstorage`. Accessed: January 5, 2018.

31 “IndexedDB.” `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB`. Accessed: January 7, 2018.

32 “Indexed database.” `https://www.w3.org/TR/IndexedDB/`. Accessed: January 7, 2018.

33 “Web SQL database.” `https://www.w3.org/TR/webdatabase/`. Accessed: January 7, 2018.

34 “Permissions API.” `https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API`. Accessed: January 7, 2018.

35 “Permissions.” `https://w3c.github.io/permissions/`. Accessed: January 7, 2018.

36 L. Olejnik, G. Acar, C. Castelluccia, and C. Diaz, “The leaking battery,” in *Revised Selected Papers of the 10th International Workshop on Data Privacy Management, and Security Assurance - Volume 9481*, (New York, NY, USA), pp. 254–263, Springer-Verlag New York, Inc., 2016.

37 “Battery status API.” `https://developer.mozilla.org/en-US/docs/Web/API/Navigator/getBattery`. Accessed: January 12, 2018.

38 “Battery status API.” `https://www.w3.org/TR/battery-status/`. Accessed: January 12, 2018.

39 “Adblock plus documentation.” `https://adblockplus.org/documentation`. Accessed: May 21, 2018.

40 “Ghostery adblocker.” `https://www.ghostery.com/`. Accessed: May 21, 2018.

41 “JavaScript/CSS font detector.” `http://www.lalit.org/lab/javascript-css-font-detect/`. Accessed: January 12, 2018.

42 “Geolocation.” `https://dev.w3.org/geo/api/spec-source.html`. Accessed: January 15, 2018.

43 “Geolocation.” `https://developer.mozilla.org/en-US/docs/Web/API/Geolocation`. Accessed: January 15, 2018.

44 “Device orientation.” `https://w3c.github.io/deviceorientation/spec-source-orientation.html`. Accessed: January 15, 2018.

45 "DeviceMotion event." https://developer.mozilla.org/en-US/docs/Web/Events/devicemotion. Accessed: January 15, 2018.

46 "Ambient light sensor." https://developer.mozilla.org/en-US/docs/Web/API/Ambient_Light_Events. Accessed: January 21, 2018.

47 "Ambient light sensor." https://w3c.github.io/ambient-light/. Accessed: January 21, 2018.

48 "Proximity events." https://developer.mozilla.org/en-US/docs/Web/API/Proximity_Events/. Accessed: January 21, 2018.

49 "Proximity sensor." https://w3c.github.io/proximity/. Accessed: January 21, 2018.

50 "MediaDeviceInfo." https://developer.mozilla.org/en-US/docs/Web/API/MediaDeviceInfo. Accessed: January 30, 2018.

51 "Media capture and streams." https://w3c.github.io/mediacapture-main/. Accessed: January 30, 2018.

52 "Web browser security." https://browserleaks.com/. Accessed: January 30, 2018.

53 "MediaStream recording." https://w3c.github.io/mediacapture-record/MediaRecorder.html. Accessed: January 30, 2018.

54 "MediaStream recording API." https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API. Accessed: January 30, 2018.

55 "Web audio API." https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API. Accessed: January 31, 2018.

56 "Web audio API." https://webaudio.github.io/web-audio-api/. Accessed: January 31, 2018.

57 "Fingerprint2 library." https://github.com/Valve/fingerprintjs2. Accessed: Feburary o3, 2018.

58 "WebGL debug renderer info." https://www.khronos.org/registry/webgl/extensions/WEBGL_debug_renderer_info/. Accessed: Feburary 06, 2018.

59 "WebGL debug renderer info." https://developer.mozilla.org/en-US/docs/Web/API/WEBGL_debug_renderer_info. Accessed: Feburary 06, 2018.

60 "HTML5 canvas fingerprinting." https://browserleaks.com/canvas. Accessed: Feburary 08, 2018.

61 K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," 2012.

62 "Princeton web transparency & accountability project." https://webtap.princeton.edu/. Accessed: May 9, 2018.

63 "Notification.requestPermission." https://developer.mozilla.org/en-US/docs/Web/API/Notification/requestPermission#Browser_compatibility/. Accessed: May 9, 2018.

64 "Notification.requestPermission." https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/NotificationProgrammingGuideForWebsites/LocalNotifications/LocalNotifications.html#//apple_ref/doc/uid/TP40012932-SW1. Accessed: May 9, 2018.

65 "Express-Node.js web application framework." https://expressjs.com/. Accessed: May 11, 2018.

66 "Let's Encrypt-Free SSL/TSL Certificates." https://letsencrypt.org/. Accessed: May 11, 2018.

67 "NGINX reverse proxy." https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/. Accessed: May 11, 2018.

68 T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," STD 90, RFC Editor, December 2017.

69 "Android asset packaging tool." https://elinux.org/Android_aapt. Accessed: May 11, 2018.

70 "Permissions overview." https://developer.android.com/guide/topics/permissions/overview.html. Accessed: March 7, 2018.

71 M. Mehrnezhad, E. Toreini, S. F. Shahandashti, and F. Hao, "Stealing PINs via mobile sensors: Actual risk versus user perception," *CoRR*, vol. abs/1605.05549, 2016.

72 "Screen." https://developer.mozilla.org/en-US/docs/Web/API/Screen. Accessed: December 27, 2017.

73 "Generic sensor API." https://www.w3.org/TR/generic-sensor/. Accessed: January 15, 2018.

74 M. Mehrnezhad, E. Toreini, S. F. Shahandashti, and F. Hao, "Touchsignatures: Identification of user touch actions and pins based on mobile sensor data via javascript," *CoRR*, vol. abs/1602.04115, 2016.

75 "WebRTC API." https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API. Accessed: January 30, 2018.

76 "WebRTC API." https://webrtc.org/. Accessed: January 30, 2018.

77 "WebGL API." `https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API`. Accessed: Feburary 03, 2018.

78 "Canvas API." `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API`. Accessed: Feburary 08, 2018.

79 "Browser market share worldwide." `http://gs.statcounter.com/browser-market-share#monthly-201801-201801-bar`. Accessed: Feburary 10, 2018.