POLITECNICO DI TORINO

Corso di Laurea in Computer Engineering

Tesi di Laurea

A Study on the Use of Mobile-specific HTML5 WebAPI Calls on the Web

Relatore
prof. Antonio Lioy

Francesco Marcantoni

Marzo 2019

Alla mia famiglia

Summary

The growth of smartphones diffusion in the last decade and the pervasiveness of the web in the current lifestyle pose the attention on the privacy and security of the users. While it is well known how browser-related data accessed during navigation can be used to harm the privacy of the user, this work aims to fill the knowledge gap concerning mobile-specific information retrieved by web-pages when visited from a smartphone.

In particular this study focuses on Firefox browser for Android devices. To detect the number of websites that have access to mobile-specific information we propose a crawler called *FFAutomator*, consisting in a Python script, that exploits the possibility to remotely control an Android device from the computer to instrument the browser and scrape the information we need from them. The script is able to open a new instance of the browser, load a website and simulate the user interaction with it. It take cares of injecting touch events corresponding to gestures on the touchscreen. We designed this program to be robust and to run for long time in order to analyze as many websites as possible. Plus, it was developed to successfully handle issues that can come up during simulation of web-navigation and that can compromise the results. An example is an unwanted redirection from the current website to an external one.

Detection is done using a proxy server to intercept *http* traffic coming to the phones and to inject JavaScript code that can log whenever a method is called or a property is read by the website and the contained *frames*. To allowed code to be run in pages it was necessary to turn off some policies that are enforced by the browser to prevent external JavaScript code to run in it.

We then elaborate the logs obtained after having crawled, through the script described before, the first 200k most popular websites according to *Alexa* ranking. Results are analyzed in a quantitative way, showing the number of websites that exploit APIs retrieving mobile-specific data and which of them are the most used. We also study the source of the JavaScript files that contain those APIs to look at the number of websites that execute external files to gather data. Given that, we differentiate the calls originated from frames and external sources from the one requested by main page.

In this study we propose also e mitigation technique, to protect who browse the web from smartphones without affecting the user-experience. This consists in an **extension** that can be installed in Firefox browser for Android that detects all mobile-specific APIs accessing data from the smartphone and allows also the users to choose to block this data retrieval or not. Plus the user can create custom rules that applies only to some chosen domains and than the default settings applying for all the other pages. The technique used to detect APIs in the extension, is the same exploited for scraping for websites. JavaScript code is injected from external sources loaded in the extension and, being the extension considered a trusted source by the websites, there is not any problem related to security policies.

Contents

Chapter 1

Introduction

Usage of smartphones among population constantly increased in the last ten years. In the United States, almost the totality of mobile phone owners has a smartphone [1]. Among all the functionalities offered by those devices, one of the most employed is **web browsing**. For the first time, in 2016, internet traffic generated from mobile devices overcame the one produced by desktop computers worldwide [2] and in 2016 56% of traffic to top-sites in the United States came from mobile devices [3]. The main factors that influenced this trend are the improvements in communication technology (e.g. the introduction of fast mobile internet connections), the development of web browsers offering now features comparable to the desktop ones and the improvement in smartphone usability (e.g. the implementation of practical gestures and the usage of bigger screens) [4].

Moreover, smartphones constantly monitor their status and are always aware of what is happening around them. This is possible thanks to the capability of those devices of retrieving information from the surrounding environment (using, for example, the ambient light sensor or the microphone) and from their status, as speed or orientation (using for example gyroscope, accelerometer and GPS). While this characteristic of smartphones enhances the experience of the user with the device, it also has a dangerous throwback: this data constantly collected by the smartphone can be gathered and used by an adversary to obtain sensitive information about the user. In the last years, an increasing concern for the privacy of smartphone users brought mobile OS developer to give the user the possibility to control which information any application can collect. Considering **Android OS**, only with version 6 of the operating system (released in 2015), the user had the possibility to grant or not, to the application, the access to pieces of information gathered by the smartphone [5]. On the other hand, only the access to most critical data (position, camera, microphone, contacts etc.) requires an action by the user while the others, including most of the one read by sensors, don't.

The combination of the increase in the usage of mobile devices to navigate the web and the quantity of data that can be retrieved by the smartphone led to new *dangers* for the privacy of the users. Browsers for smartphones are capable, like all other applications, to retrieve data from smartphones through the OS and through the **HTML5 WebAPIs** [6], websites can access most of them. This means that potentially, whenever we visit a webpage, the latter can acquire information on our device. This happens also for navigation from desktop PC and not necessary only for mobile devices, but the latter offers a wider range of data, thanks to the greater quantity of sensors they have when compared to desktop configurations.

The pervasiveness of the web in contemporary society makes many security-interested users concerned about their security during navigation. This preoccupation is justified by the huge amount of attacks that are carried by web-pages and that users can suffer while surfing the internet. A subset of them can be avoided with a careful navigation. For example **phishing** attacks require a distracted user that types his personal information in some fake website that is very similar to the real one. On the contrary other threats can harm even security-aware users. Malware can be hosted by web-server and executed when the user visit the website and loads the page. An example is JavaScript code that exploited user computational power to mine cryptocurrencies [7]. Malicious code is *delivered* to the user through different ways, it can be coded in the webpage, in third-party elements or in advertising [8]. Most popular browsers now rely on external services to prevent navigation in domain that are considered unsafe, an example is the **Google SafeBrowsing API** that gives access to a continuously updated library of websites containing malicious code.

Alongside with traditional attacks one the web, in the last years, people started developing worries about internet surveillance. The Snowden disclosures in 2013 [9] and the previous *boom* of social-networks with the habit of self-disclosing personal information, made many users aware of how, nowadays, is important to have control on personal information. Most of them ask for the possibility to remain *anonymous* while visiting websites or, at least, to leave behind the minimum amount of *evidences*. All the most popular browsers now implement **private modes** to accommodate the described needs of the users. One of the advantage of this way of surfing the web is that hosts will not be able to track the user through different sessions [10]. In order to do that, **cookies** are blocked and other data stored by the browser during the session, as history and web cache, is deleted when navigation ends [11].

Traditionally, servers keep track of connections with the hosts through the **cookies**. They are small pieces of information exchanged by two terminals when an *HTTP* connection happens and they are exploited to offer a *personalized* experience for each client. An example is the shopping cart in e-commerce websites that contains the products a user is willing to buy even if the browser is closed and reopened. In the last few years, the massive introduction of **third-party** elements in websites, made the cookies the perfect tool to keep track of the users surfing the web and profile them collecting information about their habits. The library can *recognizethe* the users visiting the first-party websites where it is implemented and consequently can infer the full history of the them.

Exploitation of cookies is not the only way to profile the users tracking them during navigation. Another one is called **browser fingerprinting** (or **device fingerprinting**). This approach consists in collecting characteristics of the machine running the browser and of the browser itself to generate e unique fingerprint of the device. Assuming that data used to build this mark makes it unique (*uniqueness*) and that does not change over time (*stability*), it is possible to know which is the machine currently navigating a given webpage [12]. Studies proved that a part of the device information can be used to reach this goal. The browser recent history for example consists in a unique fingerprint to identify an host as proved by Olejnik et al. [13] who had success in uniquely identifying the 69% of the total histories analyzed. Lot of software houses and computer companies started to develop protections against fingerprinting in their products. Recently Apple presented in the new MacOS, an updated version of Safari that should contrast fingerprinting techniques providing websites only few standard pieces of information to make the device as less unique as possible [14].

The growing usage of smartphones for web navigation made traditional techniques deployed for fingerprinting devices less effective because the pieces of information that were used to build the unambiguous fingerprint for desktops turned out to be standardized in the majority of browsers for mobile devices. Plus, most of them are not *customizable* by the user and for this reason there is a lack of uniqueness among different machines. For this reason, it is less affective to fingerprint users navigating from a smartphone using traditional device characteristics [15]. On the other hand, the development of new *mobile-specific* **HTML5 WebAPIs** offered new possibilities for trackers to exploit data that cannot be retrieved by desktop machines. A lot of studies analyzed which, among the huge amount of input collected by smartphones sensors and collected by web-servers, can result in a new opportunity to fingerprint a device. Olejnik et al. [16] studied how the precision of Battery Status API in retrieving information about maximum capacity of the battery and discharging time, made those pieces of information suitable to fingerprint users. The worry caused by this leakage made Mozilla completely abandon the API in version 52 of Firefox browser in 2017, demonstrating how dangerous the leakage of this information. Plus Mozilla showed interest in keeping who uses its products safe.

Another issue related to sensors in modern smartphones is that the high resolution of read data gives the possibility to detect imperfections caused by manufacturing of sensors themselves. It means that the biases introduced in building process can uniquely label a module and consequently the device where it is installed. Das et al. [17] prove that the combination of several classifiers strongly increases the accuracy in generating a unique *fingerprint* for the device. They succeeded in identifying smartphones through data retrieved from the **motion sensors** of the device. Furthermore, their classification technique is able to distinguish between same model smartphones, demonstrating that the possibility for an adversary to associate data readings to the device that produced them, does not depend only on the sensors implementation in each model but on the *manufacturing process* of each individual device.

1.1 Problem statement

Although many studies focused on how data retrieved by websites using *HTML5 WebAPIs* can be used in a harmful way related to users, there is not any study about how this potential privacy issue is extended on mobile devices. Even if the problem of information leaks coming from HTML5 WebAPIs has been treated in [18], there is no reference to a smartphone-related issue. That is, questioning how many websites among the most popular ones, gather information from the mobile devices.

In addition, according to the different usage of web browsers depending on the platform they are used on [4, 19] and to the different vulnerabilities and information leakages specific to mobile browsers, there are no studies on how these affect the privacy of the users.

The aim of this study is too deeply analyze how information about mobile devices is requested by websites through a **quantitative** and **qualitative** analysis. While with the first one we count the number of websites exploiting mobile-specific WebAPIs, with the second one we analyze the kind of data read to understand whether there are dangers for the users or not. Given that, we want to propose a technique that suits mobile devices to mitigate discovered dangers. Countermeasures to the threats caused by *HTML5 WebAPIs* have been developed by previous studies. Snyder et al. [18] developed an extension for desktop version of Firefox that, based on the preferences of the user, blocks the access to the selected APIs. On the other hand there is not any solution *tailored* for the smartphones and, even though many companies, as Mozilla, that develop browsers are focused in reducing risks for the users, there is currently a security flaw for who use smartphones.

1.2 Contributions

1.2.1 Analysis on websites exploiting HTML5 WebAPIs

Recently many studies focused on the capability of websites to collect data from devices used to visit them. Some of them studied how many websites among the most visited one, read device information without really needing it to work properly [18], while others pinpointed the type of harm, a given information, can cause to the user [16, 17, 20]. Considering the booming of browsing from mobile devices, and the consequent introduction of mobile-specific APIs with HTML5, we investigate how many of the most visited websites in the world exploit those APIs finding that more than 5% in the top 5000 read mobile-specific inputs from the device. We first look at which APIs actually retrieve smartphone-specific data and then we register when a website calls. The approach followed to intercept those function consists in exploiting an external **proxy server** to inject custom JavaScript code in the pages, the code *hooks* any access to the considered resources logging it into a log-file. Sites are visited from **Android** devices, instrumenting it to automatically load domains and simulating a brief navigation for all of them.

We also analyze results obtained by the scraping of all websites, looking for possible dangers for the users. Looking at well-known techniques exploiting characteristics of smartphones to track users harming their privacy, we identify how many website present this threatening behavior. Plus, we look for distribution of those harmful pages among the most visited ones, finding that the percentage of them is bigger at the top of *Alexa top 1 Million* list and decreases going down the list.

1.2.2 Mitigation technique

To reduce the risk of being tracked when surfing the web from the smartphone, we developed an **extension** for Firefox for Android, that has the double function of detecting when a website access information of the device and blocking them whenever it is requested by the user. The latter has the possibility to customize the extension creating fine grained white-lists allowing functions on websites that need them to being properly visualized. This add-on pursuits its goals by injecting JavaScript code both for the detection and the blockage. Being the first always active, the code is suited not to affect user-experience during navigation.

1.3 Overview

The structure of the thesis is the following:

- **Chapter 2:** In this chapter we describe the state-of-the-art for what concerns security and privacy of the users during navigation. We analyze the already known issues with *HTML5 WebAPIs* and we provide a background for fingerprinting techniques, being them the current most exploited technique to stealthily track users. Furthermore we give a context concerning *Android* with a focus on how the OS manages **permissions-protected** data retrieval and which information can be used to extract private information of the user. We study also the vulnerabilities found on browsers and how add-ons have been used to mitigate them.
- **Chapter 3:** This chapter contains the description of the approach used to intercept and register all the mobile-specific HTML5 in a single website. It includes the different paths we evaluated and which one resulted to be the best for our needs. We also explain the purpose of each tool we used and we describe the environment on which we conducted our study.
- **Chapter 4:** In this chapter we explain how we managed to extend our approach from a single website to the most popular ones taken by the *Alexa top 1 Million* list. We describe the algorithm behind the program we developed to automatically visit as many pages as possible, minimizing the total time needed. We particularly focus also on issues found during navigation and how we handled them to build a robust software. We also analyze how we controlled issues caused by corner-cases that required too much time and complexity to be handled.
- **Chapter 5:** In this chapter we show the results with a critical analysis of them, including statistics and meaningful patterns we found in our study.
- **Chapter 6:** In this chapter we propose our solution to mitigate the previously described threats through a Firefox extension, designed for mobile navigation, that gives to the user full control on data collected by websites. We underline here also the differences from the approach used for scraping of the information, that allowed to make user navigation experience as untouched as possible.
- **Chapter 7:** In this chapter we summarize the results obtained by the study and we provide hints for possible future works that can follow the path traced by our research.
- **Appendix A.2.3:** In the appendix we provide the user and programmer manual for the *FFAutomator* program we developed to crawl website and scrape information out of them.

1.4 Environment description

According to other studies involving exploration of the most popular websites, we retrieved them from the **Alexa top 1 Million** list. Given the huge amount of websites we had to visit, we exploited four **Android** smartphones at the same time. We flashed the version **7.1.2** of the operating system and we gained the `root` access to have full control on the device with the possibility of working at system level. Even if we operated on the top of the browser, it turned out that accessibility to the lowest levels of the system was fundamental to know if data was really read from the hardware and to solve issues during navigation that will be deeply discussed in Section 4.5. To do that, we exploit the **Xposed framework** [21]. It is a tool that can be installed only in rooted devices and that allows the user to install *modules* operating at system level and that can be developed exploiting libraries provided by the framework itself.

Regarding the control of the smartphones, to make them automatically visit websites and collect information about exploited APIs, we used **adb**. **Android debug bridge**, a command-line tool contained in the Android SDK, offering an interface to communicate with any smartphone from the computer. The mobile browser we used for the study is **Mozilla Firefox** version 59.0.1 and the motivations that lead us to this choice will be discussed in Section 3.6.2.

For what concerns the JavaScript interception in web-pages, we injected code that logs every access to properties and every call to functions we are interested in, exploiting a **proxy server**. To run the latter, we used **mitmproxy** [22], an open-source software that offers a powerful API to parse HTTP responses received by the web-server and consequently inject custom code. In particular we used the command-line version of this program called **mitmdump** because it is more verbose and because it provides more options that are essentials for our study.

Chapter 2

Related Work

In this chapter we describe the implementation of modern browsers including possible differences between platforms they run on with a focus on **Android** mobile devices. We also analyze the security issues created by those complex programs, together with the main countermeasures developed to protect the users.

2.1 Android smartphones

The Android share in smartphones market constantly grew since its introduction in 2008, reaching the 82.8% in 2015 [23]. This increase in usage, together with the huge range of services a smartphone can nowadays offer, brought to an interest of attackers in finding and exploiting vulnerabilities in this operating system.

Many studies have been conducted on the **leakage** of *user's personal information* caused by an improper usage of Android APIs in the applications. Grace et al. [24] found flaws regarding untrested apps accessing protected information from the trusted ones. Those leaked data can be used for fingerprinting the device as underlined by Enck et al. [25] that, analyzing top 1100 free apps in the offical store, found some of them collecting device info as the *IMEI*. Plus, the high number of vendors running Android on their devices, introduces substantial differences depending on the model of the smartphone. This makes Android devices easier to identify if the right information is collected. The second most popular OS running on smartphones is **iOS** from Apple. Being the latter the only vendor implementing that system in a device, it is not trivial to distinguish them using the same attributes exploited for Android phones [26].

A wide branch of research focused on this issue, developing several approaches to detect dangerous applications and to mitigate the risks for the users. Following different approaches and covering different aspects of the issue, many developers proposed solutions. Some of them perform static analysis of the taints [27, 28] while others exploit a dynamic approach [29]. The common outcome of those studies is that most of the leakages are caused by advertising libraries and that the main problem is discerning in user-intended data propagation or not, problem that is addressed by Yang et al. [30]. Seo et al. [31] extended Android permissions system to allow developers to separate permissions in-app, limiting the access of external libraries to protected APIs.

For the same purpose, the Android **permissions system** [32] was studied since it was introduced at *run-time* with 6.0 version of Android (SDK 23) in 2015. The rising concern was about third-party libraries accessing protected information without notifying the user, once the permissions were granted to the application *hosting* them. While the user can choose at *run-time* which pieces of personal information the application can access, there was interest in knowing about the origin of the methods and whether they were necessary for the core of the application to work properly. For this reason several paths were followed to keep track of protected calls invoked at the OS level, to understand first, if they were essential for the correct working of the application and second if the source could be trusted or not [33].

All those studies on taints are focused on Android applications, but similar leakages can happen also during web navigation given the wide range API system supported by most popular browsers that makes websites able to read lot of information that the smartphone acquires and elaborates. Our study aims to cover this gap taking into account also the intent of the user in disclosing personal information with the websites.

2.2 Mobile web-browsing

Browsing from a smartphone is an activity that became very popular in the last years with the development of devices facilitating web navigation even in screens that are very small when compared to the ones desktop computers. This transition was possible thanks to the introduction of *multi-touch* gestures and to the performance improvment of devices and of the mobile network [4].

This trend [2] led to the development of **mobile-specific** versions of regular websites. The experience using the smartphone is different because of the different **input** styles and of the small screen size. The differences regarding both the layout and the **functionalities** of the pages, expose the user to greater threats. As an example, studies proved that **phishing** attacks are easier to deploy on mobile websites rather than desktop counterpart and there is a big part of literature trying to *mitigate* this issue proposing detection systems for those malicious websites. This ease is given by the short address bar (because of the reduced dimension of the screen) that hides part of the URL making the discovery of *fake* domains, imitating the real one, more difficult. Plus, because of the nature of mobile browsing, users navigating from smartphones use to be less careful to details in pages that could have helped them in spotting the scam [34, 35].

Given the above listed differences, countermeasures developed for navigation from desktop browsers are often ineffective in identifying and blocking threats designed for mobile browsers. Plus, web-servers can detect if the source of navigation is a mobile device and load a dedicated version of the website with special content. Given that, Tripathi et al. [36] propose a framework to detect malicious websites designed for mobile browsing. Detection of malicious domains is done through a static analysis of the content of the webpage. The user has to type the URL of the domain in the **browser extension** that will communicate with a back-end server and will send a real-time response to the user about potential dangers. While recognizing the effort in creating a real-time solution that is suitable for all kind of users, being an extension very easy to install and intuitive to use, there is no reference to *fingerprinting* attacks that can take advantage of the mobile-specific APIs that are not available in desktop configurations. Furthermore, considering the kind of attacks that are addressed, it is enough to warn the user about the risks without granting a safe navigation of the domain.

2.3 Browser functionalities

In the last ten years, there was a constant growth in **browser** complexity, with several features being added. One of the main causes of this advancement is the expansion of the browsers to slowly substitute functions provided by native applications in different devices. Most of the browsers implement versions for different platforms including desktop and mobile reusing the same code and consequently increasing the number of lines of code in them. Vendors justify this phenomenon, that can be seen even in **browsers OS** as Chrome OS, stating that features cannot be removed in order to continue supporting websites exploiting them. On the other hand, security-concerned users and activists oppose to this trend, reporting possible flaws and proposing mitigation techniques. It was demonstrated, for example, that **WebRTC API** leaks IP address of the client. This endangers both the users navigation from a VPN and not because, even if the private IP is not considered a sensitive information, it can be used together with other characteristics to fingerprint the user. For this reason many add-ons taking care of this issue have been released for major browsers [37].

Another example is the **Battery Status API** [38], that after several attacks proving that it could be used to track the users through the uniqueness of some pieces of information about batteries that could have been read through it [16], was abandoned by Mozilla in Firefox browser [39, 40].

Many online services that are usable from mobile devices, like **social networks**, allow the user to choose between the *webpage* and the *native-app*. While exploiting the application the user can enjoy a better experience [41], web-pages guarantees a lower risk of leaking private information. For this reason a privacy concerned user should prefer using the browser to access a service instead of the dedicated app. On the other hand, browsers leak information too and to obtain a secure experience, users have to resort to mitigation techniques. Papadopoulos et al. [42], propose an Xposed module that blocks the propagation of data from Android application preventing leakages. Unfortunately, installing Xposed framework and the related modules is not a trivial operation and is not suitable for all kind of Android users. Instead, leakages prevention in web browser can be done through extensions. These software packages are nowadays easy to install through *web-markets* provided by the browser vendor and can be reached by less experienced users too.

With the introduction of **HTML5**, many functions have been standardized by *W3C* (World Wide Web Consortium) to facilitate their integration on websites and improve support by browsers. They include, for example, the access to the location of the device, the access to the microphone and cameras and the possibility to reproduce multimedia objects [43]. The improved version of HTML implemented features that were previously widely used by web-pages thanks to external *plugins* or proprietary software. An example is given by the *Adobe Flash Player* plugin, used to play media contents on the websites, that was progressively abandoned and substituted with the HTML5 *media elements* [44].

A mobile OS like **Android** provides to application developers an API to access device information and to fulfil a wide variety of tasks on the smartphone. Similarly, browsers support HTML5 WebAPIs to create a link between the JavaScript functions and the system API. This is why websites are becoming as powerful as regular applications and, for this reason, considering the several studies focused on apps accessing sensitive information, a specific study on them is necessary. APIs that we believe filling the gap between websites visited by smartphone and mobile applications, grant access to the microphone, cameras, vibration motor and all sensors that are nowadays available in modern smartphones (accelerometer, gyroscope, proximity and ambient light) [45, 46, 47, 48, 49, 50].

2.3.1 Firefox browser architecture

The browser we will take into account in this study is Firefox for Android and we will explain the reason of our choice in Section 3.6.2. Firefox for mobile devices, in particular Android OS, is based on the version for desktop platforms. It has a **layered** structure that can be seen in Fig. 2.1 [51]. The layered structure facilitates the porting from desktop to other platforms and this is the reason why the browser engine, that is the core of the browser, is the same in both Android and desktop versions. The main components of Firefox are:

- **User interface:** is the layer that makes the user communicate with the *browser engine* that is the core of the application. In Firefox the UI is built with **XUL** (XML User Interface Language), a *markup language* developed by Mozilla.

- **Gecko:** is the **browser engine** of Firefox. It is developed by Mozilla and now, in last versions of the browser, include layers that were previously separated. Now Gecko includes also the JavaScript interpreter (SpiderMonkey in Firefox) and the network module that handles secure connection. Another subcomponent is the *rendering engine* that parses the HTML document, integrates the CSS and renders the document with the related layout. Gecko is also the module that handles *persistent memory* of the browser and extensions.

- **Android OS:** the operating system is not properly a subsystem of the browser but provides lot of libraries for the user interface and manages the access to the hardware of the device.

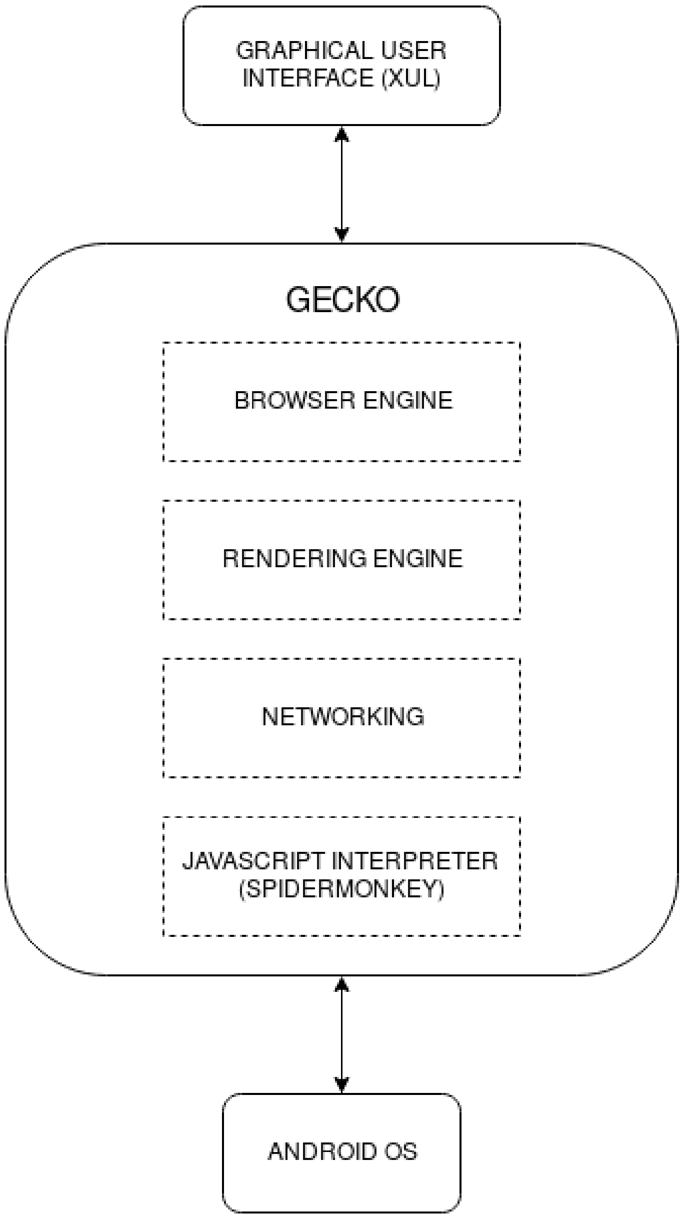

GRAPHICAL USER INTERFACE (XUL)
GECKO
BROWSER ENGINE
RENDERING ENGINE
NETWORKING
JAVASCRIPT INTERPRETER (SPIDERMONKEY)
ANDROID OS

Figure 2.1. Model of Firefox architecture on Android

Firefox OS

In 2013 Mozilla started developing a mobile operating system, called **Firefox OS**, based on the namesake browser, that inherits its main components too. It runs on a very basic Linux kernel and rendering operation is powered by Gecko [52]. Applications are substituted with *web-applications* that are developed in HTML5, CSS and JavaScript. The hardware of the device is retrieved used the HTML5 WebAPIs previously described. This was possible thanks to the introduction of APIs providing access to the hardware of the phone. This consisted in a big step toward the creation of *web-applications* that could be compared, for what concerns functionalities, with regular applications. So, even if Mozilla abandoned the Firefox OS project in 2016, other operating systems based on the web are being developed like Chrome OS by Google. This trend proves how HTML5 WebAPIs can be compared to OS system calls and how important is to ensure that websites use them in a safe way to guarantee the security of the user.

2.3.2 Vulnerabilities overview

According to recent studies, lot of those features implemented by browsers expose users to several **attacks** and well known "Common Vulnerabilities and Exposures" (CVEs). The latter are discovered bugs found in software and publicly reported. There are many researches in literature regarding the usage of new *HTML5* functionalities to facilitate attacks both for the user (*browser-side*) and for the server (*server-side*).

Considering the *browser-side* of the connection, possible attacks that are easier to commit thanks to *HTML5* features are:

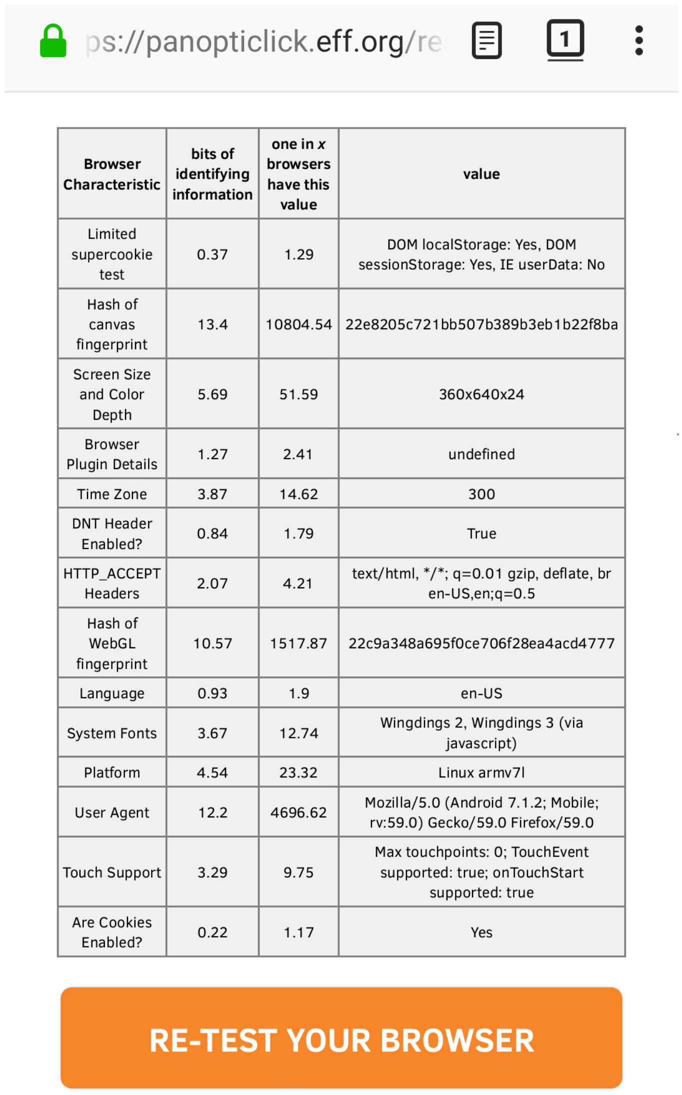
- **Cross Site Scripting (XSS):** it is an attack consisting in the injection of external malicious code in a website. When the user loads the trusted webpage, the code is executed, retrieving user sensitive data. Some tags in *HTML5* allow an adversary to inject and execute code without being filtered by common *XSS* defenses implemented by modern browsers. For example Don et al. [53] proved that the `onError` field in Media elements (introduced in HTML5), executes code when an error occurs in the media player. An adversary could insert any script in that section that would have been executed bypassing XSS security measures.

- **Browser fingerprinting:** it is a technique that consists in gathering information related to the browser and to the environment running it, including the hardware of the machine, and then using it as a **fingerprint** of the device. making web-pages capable of tracking the users. *HTML5* introduced new elements in web-pages that are well known to be used for fingerprinting as the **canvas element** and other APIs retrieving data from hardware. Fingerprinting has been subject for many studies in the last years that analyzed the techniques used to make it possible. The first comprehensive study about fingerprinting was conducted by Eckersley [54], with the *Panopticlick* project. They built a website that can be visited to discover how vulnerable the browser (and the device) is for what concerns online tracking. Part of detection is done through the open-source library **fingerprintjs2**, publicly available on *GitHub* [55]. It contains all APIs that were proved to be used to track users, but it does not contain yet any mobile-specific ones even if they planned to add support to the *Accelerometer*. Data from *Accelerometer* and other sensors available nowadays on all mobile devices, can be exploited to find manufacturing imperfections that uniquely characterize each sensor [56, 57]. We visited `https://panopticlick.eff.org/` and we tested Firefox browser from the Android device we used for testing. From the results, that can be seen in Fig. 2.2, the browser is exposed to fingerprinting attacks but there is no mention to the mobile APIs we are going to treat in this study.

2.3.3 Usage of HTML5 API

Snyder et al. [58] studied the number of features used by the most popular 10000 websites according to *Alexa* and found that more than 50% of them are never used. Plus they discovered that lot of them are most of the times blocked by **ad-blockers** and other anti-tracking extensions meaning that probably most of them are not essential for the visitor but useful for the author. This research was entirely done on Firefox for desktop and for understandable reasons does not consider mobile-specific APIs. Plus, to intercept data retrieval through methods or properties, they use a Firefox extension to inject JavaScript code in the page.

We used a proxy server instead of a plugin to make our approach suitable for any kind of browser without the need to port the extension to other platforms. Furthermore, not all mobile browsers support extensions yet. Another difference, compared to our approach, is that to control when a property (related to the considered APIs) was read by the website, the authors exploited the JavaScript method `Object.watch()`. That function allows to associate an *handler* function to a property and it is executed every time the value of that property changes. The usage of *watchpoints* was deprecated and removed by Mozilla in Firefox 58. For this reason we had to bypass this issue, hooking access to properties exploiting the **getters**. We will discuss about this later in Section 3.3.



Browser Characteristic	bits of identifying information	one in *x* browsers have this value	value
Limited supercookie test	0.37	1.29	DOM localStorage: Yes, DOM sessionStorage: Yes, IE userData: No
Hash of canvas fingerprint	13.4	10804.54	22e8205c721bb507b389b3eb1b22f8ba
Screen Size and Color Depth	5.69	51.59	360x640x24
Browser Plugin Details	1.27	2.41	undefined
Time Zone	3.87	14.62	300
DNT Header Enabled?	0.84	1.79	True
HTTP_ACCEPT Headers	2.07	4.21	text/html, */*; q=0.01 gzip, deflate, br en-US,en;q=0.5
Hash of WebGL fingerprint	10.57	1517.87	22c9a348a695f0ce706f28ea4acd4777
Language	0.93	1.9	en-US
System Fonts	3.67	12.74	Wingdings 2, Wingdings 3 (via javascript)
Platform	4.54	23.32	Linux armv7l
User Agent	12.2	4696.62	Mozilla/5.0 (Android 7.1.2; Mobile; rv:59.0) Gecko/59.0 Firefox/59.0
Touch Support	3.29	9.75	Max touchpoints: 0; TouchEvent supported: true; onTouchStart supported: true
Are Cookies Enabled?	0.22	1.17	Yes

Figure 2.2. Results of the fingerprinting test done from Firefox in Android smartphone exploiting https://panopticlick.eff.org/

Same authors [18], using the same approach of the paper described in the previous paragraph, propose a solution to prevent websites from accessing information that can be used against the user or exploiting APIs containing vulnerabilities. They developed an extension that is able to manipulate web-pages, blocking unwanted (and sometimes unnecessary) methods and limiting, at the same time, the possibility to *break* the website *ruining* its usability. This study is still done in the desktop version of Firefox and usability of websites was tested thanks to people visiting websites and describing the differences they found with or without the active extension. To avoid breaking the websites, the developed extension blocks functions using JavaScript **proxy Object**. It also offers the user the possibility to choose the APIs to block, with the option of creating custom rules for each domain.

2.4 Sensors access in Android mobile devices and related vulnerabilities

Modern smartphones contain a wide range of **sensors** that collect information about the current state of the device and the environment surrounding it. Applications have access to those sensors exploiting the OS API and most of them do not request explicit permissions by the user. While many sensors are clearly a danger for the privacy of the user (GPS, camera, microphone etc.), some of them can look harmless and they do not even require **permissions** (gyroscope, accelerometer, proximity etc.). This last group of sensors can be used through the *Android sensor framework*. Applications can listen to dedicated events that are generated by the framework every time values, read by sensors, change or when the accuracy of the sensor changes. [1]

The access to these sensors constitute a danger for the users given the several attacks exploiting that have been proved by recent studies. We already described **fingerprinting** attacks and how they can exploit data retrieved by sensors in Section 2.3.2. In addition, other privacy violations reported in recent studies, are:

- **Activity recognition:** many studies proved that sensors can be used by an adversary to infer **private** information of the users. It is possible to deduce for example the *activity* of the smartphone owner, so if it is walking, running or in a transportation vehicle exploiting the **motion sensors** and the GPS. This is done through a classifier and does not rely neither on historical patterns nor in previous training of the users habits [59].
- **Inference of input on touchscreens:** information from motion sensors, that include the accelerometer and the gyroscope can be used to predict what the user is typing on the touchscreen of their smartphone. This is possible because, when typing, depending on the position on the screen, the orientation and the speed of the smartphone changes in a peculiar way. Xu et al. [60] developed a malicious application that was able to retrieve (using this technique) the password used to unlock the device and the sequence of numbers typed on the *dialer*. Porting the study in the browser environment, it is trivial to retrieve the layout of the current page and so infer the activity of the user.
- **Location detection:** a study from Han et al. [61] demonstrates the possibility to infer the position of a vehicle in a city through the motion sensor. In the proof-of-concept, no permissions to access position were granted by the authors and the position of the phone was correctly inferred with an high precision.

The described exploits of unprotected sensors to infer sensitive information, caused the development of defensive systems. Bai et al. [62] face the problem with an approach focused on the application. They instrument the original APK (Android Package), to make it communicate with a *policy controller*. The latter directly modifies the values generated by the events and allows the user to choose about *turning off* the sensors or randomizing data generated. This approach can be applied to browsers too but it requires a *recompilation* of the application and is not a usable path for an average user.

2.5 Browser extensions

In the previous sections, we described several threats for the **privacy** of the users navigating the web using a desktop or a mobile device. The most common countermeasures developed during the last years to address this problem concerns the creation of **browser extensions** that, with different approaches, try to address common issues without compromising the user experience. Extensions are widely supported by most common browsers for desktop platforms while this is not true for mobile OS. As an example, Google Chrome for Android does not support

[1] Android Sensors: `https://developer.android.com/guide/topics/sensors/sensors_overview`

extensions yet, same as Firefox for iOS. Extensions are used to enhance features of the browser and, for what concerns Chrome, Firefox, Opera and MicrosoftEdge, rely on similar APIs to enhance compatibility. It means that after having developed an extension for Firefox, porting it to another browser, among the previously cited, is relatively easy. [2].

Extensions developed to improve security during navigation has the goal of limiting tracking capabilities of websites. They usually implement two main functions:

- **blocking:** extensions are used to block content on the page that can be dangerous for the user, usually *stopping* requests coming from **tracking scripts**. Those are scripts used by third-party libraries to provide services to the first-party website. Those services include analytic, advertisements, social-network integration and more. Being the same library present in several websites, it can identify the user through cookies or *browser fingerprinting* to infer their navigation profile and thus personal information.
- **detecting:** extensions communicate the user whether a website is using some well-known dangerous service, giving information about it and how it can be exploited from malicious usage.

Merzdovnik et al. [63] study the effectiveness of most popular extensions in identifying and blocking trackers from third-party libraries. They conducted the research on browsers from desktop and on smartphones, taking into account libraries in Android application, but there is no reference to navigation from mobile phones. Results show that most of trackers are blocked by the most popular extensions while there are few problems with minor libraries and some fingerprinting scripts. On the other hand these extensions contain flaws regarding the detection of the trackers. All of them rely on a manually updated static black-list containing the domain of the well-know trackers that are blocked based on that. Given the dynamic nature of those libraries, the list of domain is always outdated, endangering the users. Plus they take into account only domains related to tracking companies, so user data retrieved from non-tracking-domains is not addressed. Starov et al. [64] developed *PrivacyMeter*, an extension that calculates on-the-fly a privacy score for each visited website based on privacy threatening features retrieved from the website. The *HTML5 APIs* considered by the last described study, to identify possible dangerous websites, do not include any mobile-specific function and the extension is not tested on any smartphone.

[2]Firefox WebExtensions API: `https://developer.mozilla.org/en-US/Add-ons/WebExtensions` Chrome extension API: `https://developer.chrome.com/extensions`

Chapter 3

Approach

The study consists of three main steps. First, we selected the mobile-specific functions among all the ones introduced with HTML5, filtering the list available at `mobilehtml5.org` [6]. Then, we found a procedure to intercept the selected JavaScript methods when called by websites in the mobile browser and we tested it on an expressly created *dummy* website containing all the considered APIs. After testing that, we verified that for each call in the browser the corresponding Android system function to retrieve data from the device is launched. Last, we automated this procedure making the smartphone autonomously visit all websites from the *Alexa top 1 Million* list and save the log for each one containing information related to the APIs called.

3.1 HTML5 Mobile Functions selection

First of all, we identified the functions used to retrieve mobile-specific data among the *HTML5 WebAPIs*[6]. We considered functions as **mobile-specific** if they can be always used in a mobile device, while can be useless in desktop configurations. For instance, the microphone is always available in smartphones while it is not included in all computers (e.g. most of the workstations do not have any microphone when bought). The WebAPIs we considered communicate with the smartphone in two different ways. First, the method is explicitly called and the communication with the device happens in *one shot*. Second, through events, means that properties of a class are managed by an event listener. This happens when data is retrieved by sensors that continuously fire events to update the system. Given that, to be more clear, JavaScript methodologies that will be considered in this study are divided into two groups. One shot methods:

- **Geolocation:** in `Geolocation` interface there are `getCurrentPosition` method which retrieves one-shot position of the device and `watchPosition` method. The latter is similar to the previous one, but now a process to continously watch the location is launched and its PID is returned.
- **Vibration:** method `vibrate` from class `navigator` makes the phone vibrate with the possibility to customize the length and the intensity of the vibration.
- **Media Capture:** in `Navigator` interface the method `getUserMedia` is used to access the camera, the microphone or both.

Events to retrieve data are handled setting a listener and specifying the name of the event that has to be managed. They are:

- **Motion Sensor:** `deviceorientation` event to get the orientation of the device and `devicemotion` event to retrieve the acceleration along the three axes.
- **Screen Orientation:** `change` event to retrieve changing in screen orientation (portrait or landscape).

- **Proximity:** `deviceproximity` event to measure the distance between the device sensor and any other object close to it.
- **Ambient Light Sensor:** `devicelight` event to get information about ambient light in the environment.

Given those two substantial differences in the way we retrieve data, it was necessary to find the best strategy not to miss any of them. It was straightforward that we needed a way to hook into JavaScript methods, but we had to find a solution for the events. The browser, by default, listens to all events coming from Android and whenever it happens, it fires the corresponding event propagating the flow of information. Now, in our case, the targets for the events fired by the browser were the `screen.orientation` object, for the event signaling that orientation of the screen changed, and the `window` object for all the others.

There are two ways to instantiate a listener in JavaScript. First, using the `addEventListener` method on the target object passing as arguments the name of the event we want to listen to and the callback function called when events are caught. Second, it is possible to directly associate a function with the event property of the target object.

The first case can be managed as a normal method, intercepting `addEventListener` and checking that the first argument corresponds to one of the events we want to control. The second must be managed in a different way, and it is necessary to implement a hook to register whenever a property is accessed. We will talk about this in Section 3.3 related to the injected code.

3.2 JavaScript Calls Interception

The goal of the study is to establish whether a website can retrieve data from the device through browsers HTML5 functions. First of all, we had to decide how it could be done. There are two different approaches that can be followed to hook HTML5 methods. First is hooking the function directly in the browser, intercepting methods that are called in the application consequently to a given JavaScript method. Second is to hook externally the JavaScript functions at webpage level injecting code that is capable to catch them and execute some code that logs it.

3.2.1 Native code interception

We followed this approach on the Android version of Firefox. The browser functions linking the code executed in the web pages with the methods from the **Android API** to access the device is done through **native libraries** written in *C++* and integrated into the Java code. The goal is to intercept the Java functions of the Android API and read the stack-trace, finding the link to the HTML5 functions in the document. A *stack-trace* is the list of all the memory frames allocated in the stack by routines called at a certain time of a program execution. Through that, it is possible to start from current function and then go up until the first of the "nested" procedures that originated the one taken into account.

The issue we faced is that the *stack-trace* ends with the generic Firefox thread without reaching any JavaScript routine. At this point, we downloaded the source code and, reading it, we realized which are the native functions used. Given that, those procedures cannot be intercepted as Java calls, indeed we used Frida [65], a tool that after being installed in the device, and after being instrumented by the user, is able to hook the desired native functions. Even using this tool it was impossible for us to register when those native functions were called. The reason is that because of the sensitivity of data that can be accessed through them, the function calls are *obfuscated* to prevent malicious software from reading delicate information.

To make interception of those calls possible, it is necessary to recompile the whole browser without hiding those function calls and then use Frida to hook them. This approach is difficult to follow because of the complexity of the browser and because it would introduce substantial differences with the version commonly downloaded by users. Another reason that made this approach weak is that browser is a very complex program and it may use other ways to get data from the device without using the native libraries. For instance, it may use some sort of **cache** to store data retrieved by the phone and use it the next time it is requested. In this case the JavaScript function is called while the corresponding native call is not.

Summing up, this approach does not ensure that all websites exploiting mobile-specifc WebAPIs are found because the correspondence between the JavaScript function and the calls of the browser is not guaranteed.

3.2.2 HTTP response interception

JavaScript functions are intercepted using a JavaScript code injection handled by a proxy server. Each response coming from the server hosting the website passes through a **Proxy Server** that injects in the document some proper JavaScript code that is used to hook the functions and to print a message in the system log every time an interception happens. In this way what we check is whether or not a JavaScript method is called, and, being the interception at **page level**, it is independent of how the browser manages them. For this reason, this approach can be even extended to any kind of browser.

Even if this approach seems pretty straightforward, it creates issues related first, to the execution of external code in a page that can be blocked by security measures enforced by browsers and second to a possible interference of injected code with the page that may alter its execution. Those issues will be discussed in Section 3.4.2 that describes how the code injection through the proxy server happens.

3.3 Injected Code

Given the two different of ways used to gather information, described in Section 3.1, two different hooking strategies are required. One to intercept methods and another to check whether events are accessed. Given the difference between the two approaches, we decided to use to separate scripts injected together in each website.

3.3.1 Hooking methods

To hook methods of an object we used a Node.js module available on GitHub called javascript-hooker [66]. This script allows to hook any JavaScript function called in a webpage, passing as arguments:

- **object:** the object containing the method;
- **method:** string with the name of the method;
- **function:** function called before the execution of the hooked method.

The script creates a *wrapper* around the functions and makes the wrapper substitute the original method. Given our needs, we print a message in the log when the function is called, then the original function is executed and at the end, it returns the original return value. In this way, we reach the goal of minimizing the effect of injected code in the original document. In the *wrapper* we can access the arguments passed to the original function, thanks to that we can intercept the `addEventListener` and then check if the first argument corresponds to any of the events we are interested in and generate the proper string in the log only if the correspondence exists.

3.3.2 Hooking properties

For what concerns the listening to events associating a function to the specific target property, as described in Section 3.1, we need to intercept the **setter** of the property. Thanks to the

`Object.defineProperty()`, it is possible to substitute the setter of a property adding logging code but then there is no way to set the original value and the page will not work as expected (the page does not listen anymore to the events which target was modified).

To avoid *breaking* the website, we followed a strategy seen in Chameleon [67], an open-source browser extension available on GitHub. The approach consists of overwriting `getter` for properties of each Event prototype, instead of substituting the setter relative to the target property. In this way, every-time the property is read, our custom function is called. Following this strategy, the website script execution is altered anyway. The reason is that the original value that the website expects to be returned from accessing the property, can be read only before the new *getter* is set, otherwise, the script enters in an infinite loop.

While this is not a problem for properties that are supposed to remain the same during navigation on the website (e.g. properties related to display characteristics), it can be a problem with data retrieved by sensors because it is supposed to change several times during a regular navigation in a website. The solution adopted to limit problems related to that is making the returned data *compliant* with the expected ones. For example, data regarding orientation retrieved by **gyroscope** is a decimal number in a range between −365 and +365.

Another way that could be used to hook functions directly assigned to the *Event* property of the target *Object* is to check if there is any function assigned to it after the page is completely loaded. This result can be obtained making the document wait for the page to be fully loaded through `window.onload()` event handler and, only after that, check the value of the aforementioned target property. This approach will not be implemented because, given the lack of interception, it is not possible to retrieve the *stack-trace* of the function and consequently the source file containing the function that, as described in Section 3.3.3 is an information we need to proceed with elaboration on the results.

Summing up, data retrieved through events, is handled in two different ways:

- listening to `addEventListener` on the target object checking the argument matches with the desired event;
- defining new **getters** for the properties of the events prototype.

3.3.3 Getting source of the JavaScript file

Every time a website uses one of the function we are interested in, it is important to know the **URL** of the JavaScript file executing it. This information can be used to check, for example, if the script belongs to the same **domain** or to an external one and identify any possible association between the usage of these API calls and a domain. To register the calling source we use the `stack` property of the `Error` object in JavaScript. We implemented a function called `getScript()`, declared in both the injected scripts that creates an `Error` object and then reads its `stack` property. The latter is a string containing the list of functions (ordered from the most recent to the initial global scope call) separated by a *new line* character. Each element contains the name of the function and the source, separated by a `@`.

The number of the functions called before the searched one is fixed because we exactly know how many routines are necessary to intercept a function. So we access the list of functions in the proper position and then, manipulating the string, it is easy to retrieve the URL of the source file that is saved and printed in the log as it is described in Section 4.3.

3.4 Proxy Server

Considering the approach we decided to follow and described in Section 3.2.2, we need to create a **Proxy Server**, set up as a *man-in-the-middle* that intercepts all the traffic exchanged between the device and the web-server.

A proxy configured in this way is called **Transparent Proxy** and acts like a *gateway* for the devices. To intercept in and out communications is necessary to manually configure the **IP address** of the proxy in each device. After this setup the server will forward all the **HTTP requests** to its default gateway granting a connection for the host. Clearly all the corresponding responses will go through the server as well and it is in this case that the proxy is supposed to inject the JavaScript code.

We have implemented the Proxy Server in the same private network where the smartphones were connected using the open-source software `mitmproxy` [22]. This program, through the `mitmdump` command, offers a powerful Python scripting API to control the behavior of the server and to instruct it on how to modify traffic passing through it. Plus, this software has his own certificates that can be downloaded from the official website and manually installed on the device. In this way it is possible to make connections to the websites trusted by the smartphones exploiting the HTTPS protocol. This is crucial because many of the websites that should be analyzed enforce secure connections and traffic can be decrypted and tampered (with our JavaScript code) only if the connection to the proxy is trusted.

Furthermore, `mitmdump` provides other useful options as `anticache` that prevents the proxy from sending a *304 - PAGE NOT MODIFIED* response, forcing it to retrieve always the page from the web-server and `ssl-insecure` that allows connections even if the upstream server does not have a valid TLS certificate. Thanks to this configuration we make our proxy server as transparent as possible avoiding any unwanted interference with web navigation.

3.4.1 Buffering prevention

HTTP is an asymmetric communication protocol that connects a client to a server. Clients send *REQUEST* packets to the server and the latter get back sending a *RESPONSE* containing the content of the web page. Each of these packets is composed by a **body** usually carrying the content of the page and a **header** that instead contains *meta-data*. The latter contains useful information for client-server communication as the size and the date of the file contained in the payload. One of those fields is named **content-type** and contains the kind of data carried by the packet in the body. It can be an HTML document, JavaScript code, images etc.

The first problem we ran into was that by default `mitmproxy` stores in a *buffer* both headers and bodies of all HTTP responses. When the visited website contains images, videos or heavy files in general, the proxy downloads them before sending the response to the device. In this way, loading process got stuck until all responses are downloaded and the average time spent on each website can be very long.

To avoid that, given that we had to inject the JavaScript code in HTML documents and that the **header** of each response is always buffered, we make the `mitmproxy` check first whether the **content-type** field contains data related to HTML (identified by `text/html`) or not. If so, the `stream` attribute of the response is set to `False` and as in the default case, the whole response is buffered in the proxy, ready to be manipulated, otherwise it is set to `True` and the response body is directly sent to the device preventing the server from becoming a *bottleneck*.

3.4.2 Injection

Python API for `mitmproxy` allows controlling the flow coming from the server to the device parsing the HTTP response, giving us the possibility to manipulate the desired portion of responses. Once the content is processed by the proxy, it checks that the body contains the HTML document and the latter is parsed. Now a new HTML Script tag is created and inserted at the beginning of the `head` section of the document.

All the code is directly injected in the `head` of the document (if there is one) or in the `body` otherwise and is not loaded from an external source. This way to execute code in HTML document is called **in-line scripting**. Thanks to this, we eliminate the time needed from the website to download the script and the code is executed immediatly when it gets parsed from the browser.

For this reason, we can be sure that the *hooking functions* are loaded before any other method, that must be detected, is called by the page.

On the other hand, to protect users from **code-injection** attacks, browser enforces rules called **Content Security Policy (CSP)**. However, web-servers communicate to the browser whether or not enforcing those rules through the **Content-Security-Policy** field contained in HTTP *RESPONSE*. To be sure that the in-line injected script is correctly loaded without being blocked, all the security tags related to **CSP** are stripped from the response header by the proxy through the available HTTP manipulation API.

3.5 System calls interception

We described in Section 3.2.2 that part of the process consists in verifying that for each JavaScript call there is an associated system call. We have already implemented a system to detect JS calls so, for the others we exploit the Xposed framework, installing the **PermissionHarvester** module, developed by Diamantaris et al. [33].

The latter intercepts at run-time all *permission-protected* functions writing them in the *log* with a full stack trace. Even if this module intercepts many of the calls we are interested in, it is not enough because all the functions used to retrieve data from sensors are not *permission-protected*, as described in Section 2.4 and, for this reason, they are not taken into account by the previously described module.

To integrate data gathered by **sensors** we searched which are the functions called whenever a sensor is accessed by the OS and we consequently developed a custom Xposed module to hook them [21].

Android applications cannot directly read the current value of a sensor but they have to *register* a listener for it and consequently read the captured events. Each sensor can be obtained calling the `getDefaultSensor()` method on a `android.hardware.SensorManager` object. The latter must be previously declared specifying the name of the sensor while with the `getDefaultSensor` function, a `Sensor` instance is created (this happens only if the corresponding hardware exists, otherwise it returns `null`). Then, a listener must be registered for each sensor calling the method `android.hardware.SensorManager.registerListener()` passing as one of the arguments the `Sensor` object previously instantiated.[1]

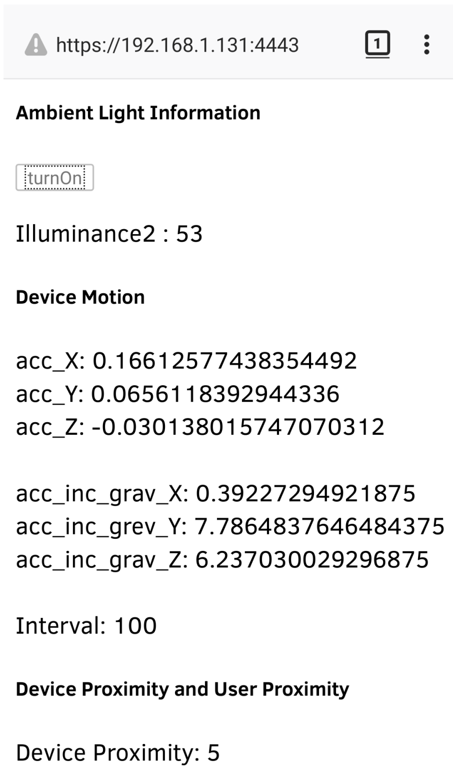
Xposed provides a library with functions that can be used to hook *non-native* methods available in any package installed on the Android device, and then execute custom code before or after them. The function we exploit is `findAndHookMethod()` passing as arguments: the name of the function, the related package, the type of the arguments that the function we want to intercept requires and the callback function that we want to be fired after the method is *hooked*. We use this procedure to intercept both methods, `getDefaultSensor()` and `registerListener()` printing on the logs the related stack trace.

To make interception possible, it is necessary that all arguments correspond to the ones effectively called by the application otherwise the intercepting method will not recognize the function. Considering that the arguments of `registerListener` function can vary depending on the options the developer want to specify, our module needs to *hook* all different combinations of values passed to avoid the risk of *losing* them.

3.6 Preliminary Tests

After having identified the functions we want to log, and after having decided the best approach to do that, as described in Section 3.1, we started doing some tests using an Android device to

[1]More information on sensors in Android available at https://developer.android.com/guide/topics/sensors/sensors_overview


https://192.168.1.131:4443

Ambient Light Information

```
Illuminance2 : 53
```

Device Motion

```
acc_X: 0.16612577438354492
acc_Y: 0.0656118392944336
acc_Z: -0.030138015747070312

acc_inc_grav_X: 0.39227294921875
acc_inc_grev_Y: 7.7864837646484375
acc_inc_grav_Z: 6.237030029296875

Interval: 100
```

Device Proximity and User Proximity

```
Device Proximity: 5
```

Figure 3.1. The interface of the website used to test the correctness of the followed approach

evaluate the effectiveness of the methodology described in previous sections. The goal of those testings was to check if the approach we had designed was correct and to help us in choosing which was the best mobile browser to conduct this study on.

Among browsers available in the **Google Play Store**, we filtered the ones with more than 100,000,000 and then, among the remaining one, we took only the ones having enough support at HTML5 APIs according to the website we consulted to obtain them [6]. Among all of them, the ones with a wider support for those APIs are **Google Chrome** and **Mozilla Firefox**.

The first tests were done on a *dummy* website. It was running on a local web-server, powered by **Node.js**, and was developed expressly to test our strategy. It is structured with an `index.html` file to create a very basic front-end interface that prints data retrieved by the device and that gives the possibility to call a function after *tapping* a button in order to better analyze what happens when a call is done. JavaScript code is in a separate file called `webapis.js` that contains the methods we are willing to intercept on public websites. A screenshot of the website running on the device used for testing can be seen in Fig. 3.1.

During the study the *demo* website was useful for several reasons:

- we could check whether the functions were actually supported by the browser;
- we could control if our script could actually hook into functions and properties access;
- we could verify that proxy server and code injection was working properly;
- we could check if the Android API was actually used by the browser to get data requested by the website.

To make this website as similar as possible to a real one, we created a *self-signed* certificate, used to make our web-server trusted by browsers. It was necessary because both Firefox and Chrome prevent, by default, websites not showing a valid certificate from accessing some of those

APIs like the one to retrieve GPS location. In this case, it would have been impossible to test functions belonging to *Geolocation API*. The local web-server listens to two different ports, one to serve the client with an insecure connection and the other to enforce the secure one.

When visiting a website we would take into account all calls done from the website, including JavaScript files contained into **iframes**. Those elements in the HTML documents are used to represent, given the URL, an entire web-page inside the first one. Our approach automatically hooks methods even in the *iframes* because packets containing them go through our proxy server that treats them as regular HTML pages, injecting code into them too.

3.6.1 Results

Together with the *demo*, we visited also other websites offering features exploiting the APIs we are interested in, to spot possible issues with our approach, but we did not find any issue. Those tests confirmed also the information about browser support to the *WebAPIs* that we acquired from `mobilehtml5.org` [6]. Plus it proved that injection of code using a *man-in-the-middle* proxy works fine both for secure connections and not and that, after visiting websites several times, the *injected code* is always executed before the code belonging to the page. Furthermore, whenever visiting websites containing *iframes* all the calls are regularly recorded and it is possible to identify them thanks to the portion of the log containing the source file.

3.6.2 Why Firefox

Preliminary testing phase was executed on both browsers (Google Chrome and Mozilla Firefox) keeping the logs. In both of them debug logging could be enabled in the *settings menu* and the logs could be filtered by the Android system logs through specific *tags* that are **chromium** for Google Chrome and **GeckoConsole** for Firefox. The hooking system, being inside the web-pages and not browser-dependent as explained in Section 3.2.2, works fine for both browser (taking into account the supported methods only described in `mobilehtml5.org` [6]).

Besides that, we looked at functions of the Android API called by the system to provide data to the browser and we checked the correspondence with the related system functions. What we did was visiting the dummy website, calling the JavaScript methods and see what was registered into the log filtering only the functions related to the system. It would be possible after that to check which are the functions used by the OS to communicate data to the browser.

While it was possible to find this link on Firefox for all the considered JavaScript functions, it was impossible to find a pattern in **Google Chrome**. For the best of our knowledge, this happens because Chrome relies on **Google Play Services** to access the information managed by them. They are a collection of individual services introduced by Google introduced in 2012, designed to limit dependencies of application from devices. They are an *interface* that continuously communicates with the hardware for retrieving and storing information. Given that, they provide a unified *client interface* that applications can use to gather data they need. Considering the method that we are using to get Android calls and that is discussed in Section 3.5, there is no possibility to hook into them and is not even possible to intercept communication between the Play Services and the interface in the application.

On the other hand, Firefox does not use Google Play Services, but data is directly read through the Android API calling the functions that can be read by logs. This is the main reason why we continued our study focusing only on Firefox.

Chapter 4

Web Navigation Automation

Once the environment was all set we started our large-scale testing, visiting the most popular websites taken by the **Alexa top 1 Million** list. In this chapter, we discuss the methodologies used to programmatic visit them and save the logs of each website organizing the results to be automatically elaborated. For each website, the interaction of the user must be simulated so that each website is *navigated* to discover even functions that are called only when a sub-page is reached. Everything is managed by a program we implemented and that for sake of simplicity we will call **FFAutomator**.

Given the huge amount of websites that must be visited, we used four smartphones, all of them with the same Android version installed (7.1.2) and with the same Firefox version (59.0) in order to have results unbiased by the phone they were retrieved from. Three of them are LG Nexus 5X and the other one is a OnePlus One.

4.1 Interfacing smartphones with the computer

The smartphones used for testing must be controlled by the computer that *acts* like a user sending commands to them and at the same time collects the information contained in the *device log*. The Android SDK offers a command-line tool called **adb** that offers a complete interface to communicate with the phones.[1] This tool instantiates a *client* on the computer for each connected device (identifiable thanks to the serial number of the phone). Each client sends commands to the *adb server* which is started on the computer after the first run and that is in touch with the adb daemon (adbd) running in the background on each device and that effectively executes commands on it.

Having several devices connected at the same time at the computer, it is possible to select which one is affected by the `adb` command using the `-s` option followed by the device *serial number*. The `adb` functionalities we used in our study are:

- `adb logcat`: prints on the command line the current complete *log* of the device with the possibility to increase verbosity, printing, for example, the date and time of each log.
- `adb shell`: creates an interactive *shell* on the phone. This command can still be used in a non-interactive way sending *single* commands. This function opens to a variety of requests that belong to *Unix-like shell* (e.g. `kill` to send signals to a process) or to the phone-specific shell (e.g. `am` to start the activity of an application). Moreover, is possible to run any command with *super user* rights placing `su` before the command string.

[1] https://developer.android.com/studio/command-line/adb.

On the other hand, for what concerns communication between the computer and Firefox, there is not any *interface* that can be exploited. For this reason, any task that the browser should do (e.g. close a navigation tab or clean the cache) has to be executed simulating the *gestures* of a user. There are several techniques to do that and we will discuss them in Section 4.2. Consequently, there is no way to make Firefox send messages to the client. For this reason, the only possibility for the *FFAutomator* to be aware of what is happening in the application is reading the log of the latter.

Clearly, this methodology opens to several issues for what concerns `synchronization`. Logs in Android are stored in four different *buffers* (depending on which part of the system originated them) before being written in the system files. For this reason, depending on how much the system is exploiting the buffers, there may be an unpredictable latency between the origin of the log and when it is readable through the `adb logcat` command. Techniques adopted to mitigate this issue are described in Section 4.5.

4.2 Instrumenting Touch Gestures

As explained in Section 4.1, the only way to control what happens in the Firefox application on the device is through the simulation of *touch gestures* and *taps*. These interactions are needed for two different aims:

- simulating user navigation on each website randomizing gestures that are commonly done by a user for navigation;
- interacting with the options and the menus of the browser.

According to Keinanen at al. [68] more than a half of the gestures used in web-navigation are **single taps** and **swipes**. Other multi-touch interactions, as *pinch-to-zoom*, are used for the convenience of the user when reading small texts or tapping on small buttons but they are not necessary to fully navigate a website and excluding them from the simulated touch events does not limit the ability to visit all sub-pages of a domain.

4.2.1 Interaction with the device

The most important requisite of the commands used to interact with the device is that the type of gesture and the coordinates where it happens must be totally under control of *FFAutomator* because they have to recreate a well-known chain of actions to obtain a result.

input The adb shell tool offers the possibility to send touch signals using the command

```
$ adb shell input <type_of_gesture>
```

where *type_of_gesture* can be a `tap`, a`swipe` or the *code* corresponding to the key available on the device. Depending on the gesture, it requires other arguments. For a *single tap* it is necessary to add two integers corresponding to the coordinates of the screen where the *tap* should happen or, for the *swipe*, two couples of coordinates are requested. They represent the starting point and the end-point of the gesture, plus an integer indicating the time length of the scroll expressed in milliseconds (ms).

The input command of the Android shell is a Java application belonging to `com.android.commands` package. [2] It works at high level in the OS, and two executions of it cannot be executed back to

[2]Code available at: https://android.googlesource.com/platform/frameworks/base/+/android-7.1.2_r36/cmds/input/src/com/android/commands/input/Input.java

back because of the substantial time frame (1-2 seconds) elapsing between them. This amount of time is not acceptable both for navigation and interaction with browser given our need to use the minimum amount of time for each website, to run our tests on as many websites as possible. Using input program, the touch events are injected at Java level communicating with the InputManager API. [69]

sendevent Another way to inject user gestures into the device is, instead of operating at *high-level* (Java level), manipulating the system at *low-level.* Android OS is based on **Linux kernel** and inherits from it lot of features at system-level [70]. Input management is an example of this inheritance. When the user interacts with the touchscreen, the latter generates events that are written in a *device file* in /dev/input folder. Then, they are decoded by the drivers in the Linux kernel and sent to the higher level of the system. [3]

Events are sent to the kernel with a standard format consisting in four fields:

```
<device_file> <type> <code> <value>.
```

It is important to clarify the difference between an *event* and a *gesture*. The first is a single interaction with the hardware while the second is a chain of events that is meaningful for the system and that recreate a gesture.

As an example, this is a series of events corresponding to a *single tap* on the screen, that can be retrieved at run-time thanks to the `getevent` command in the Android shell:

```
/dev/input/event0: EV_ABS ABS_MT_TRACKING_ID 00001006
/dev/input/event0: EV_KEY BTN_TOUCH DOWN
/dev/input/event0: EV_KEY BTN_TOOL_FINGER DOWN
/dev/input/event0: EV_ABS ABS_MT_POSITION_X 000002ec
/dev/input/event0: EV_ABS ABS_MT_POSITION_Y 00000434
/dev/input/event0: EV_ABS ABS_MT_PRESSURE 000000c0
/dev/input/event0: EV_ABS ABS_MT_TOUCH_MAJOR 00000005
/dev/input/event0: EV_ABS ABS_MT_TOUCH_MINOR 00000003
/dev/input/event0: EV_SYN SYN_REPORT 00000000
/dev/input/event0: EV_ABS ABS_MT_TRACKING_ID ffffffff
/dev/input/event0: EV_KEY BTN_TOUCH UP
/dev/input/event0: EV_KEY BTN_TOOL_FINGER UP
/dev/input/event0: EV_SYN SYN_REPORT 00000000
```

It is possible to see that the **device file** corresponding to the touchscreen is `/dev/input/event0`. This chain of events is the same for all the gestures of same category. The parameters changing are:

- the tracking ID at line 1, that is a parameter used in *multi-touch* devices to handle different touches on the screen at the same time;
- the coordinates on the screen, at line 4 and 5;
- the pressure of the finger on the screen at line 6.[4]

The `sendevent` command, available in Android, allows injecting an event at the time. Sending the right sequence of events, back to back, it is possible to inject entire gestures. The requested arguments are the same retrieved by *getevent* but they must be decimal numbers instead of hexadecimal. To find the exact chain of events needed to recreate a gesture we run the *getevent* while doing the desired action on the smartphone. The shell saves in this way the *dump* of

[3]Android input documentation: https://source.android.com/devices/input/#input-pipeline

[4]More information on the events are available in the official documentation at https://www.kernel.org/doc/Documentation/input/event-codes.txt

the generated events that, after being converted to decimals, can be used as parameters. Thus, modifying the parameters regarding the coordinates on the screen, it is possible to recreate the action we want on the device.

Unlike the input Java program, sendevent is a C program and the shorter execution time makes it more suitable for our needs.

4.2.2 Simulating navigation

The aim of the *FFAutomator* is to **scrape** information from websites simulating the user navigation. To do that, we had two possibilities:

- use tools designed to do **fuzzy testing** on Android apps;
- use one of the input simulation described in Section 4.2.1 to send gestures with random coordinates.

Monkey tool The monkey is a command-line tool included in the Android SDK that helps the developer in testing application on devices with the technique of **fuzzing**.[5] The latter is a procedure used in software testing that consists in stressing the software sending a stream of inputs while logging its behavior, trying to discover as many bugs as possible. We can exploit this technique to *stress-test* the websites trying to execute as many JavaScript scripts as possible.

The monkey tool is a pseudo-random generator of events thus is possible to set a seed to recreate always the same sequence of inputs but it is still possible to send anyway random gestures every time, that is what we wanna do given that we do not want to discover any bug but only to *explore* the content of websites. Other options of the command allow to select the target application, that in our case is Firefox browser, the number of events that the *monkey* injects and even the delay between a gesture and the following one. This is important to give to the website the possibility to load a possible new page or to render a part of the current one after scrolling. It is also possible to control the kind of the injected gestures. We can exclude, for example, interactions with the buttons of the smartphone (e.g. the volume rocker) or with the navigation bar of Android. On the other hand, the control we have on the tool is limited to the options described above and it creates the following issues:

- There is no way to control in the tool the ranges of coordinates that the *Monkey* uses. Given that, touch gestures are not confined to the web page and they can end up in the menus and interfere with the navigation. It is possible to mitigate that creating a *dummy* application that runs in the background creating a fictitious activity over the areas that must not be touched by the *FFAutomator*, that catches all the touch events. Even if we tested this approach and it was working we noticed unexpected behaviors of the browser given by interaction with "prohibited" areas.
- Taking into account the differences between *events* and *gesture*, that is explained in Section 4.2.1, the *Monkey* sends a stream of random events that compose e gesture. For this reason, there is no possibility to isolate single gestures. Having the possibility to send single actions is very important because allows checking the response of the website to them. We will describe this aspect in Section 4.5.

Randomizing single input It is possible to simulate the user navigation by sending a sequence of single gestures using one of the methods described in Section 4.2.1, generating random coordinates every time. We tried that using the `sendevent` command for the reasons described in Section 4.2.1. What we had to do is call the `sendevent` method changing the arguments related to coordinates.

[5] Documentation can be found at `https://developer.android.com/studio/test/monkey`

To simulate swipes we proceeded in the same way we did for *taps* and we realized that swipes are composed by repetition of same events, so we created a “module” and clearly, in each module, their coordinates get closer to the end-point of the action. We can consider each module as a step of the pointer from the starting point to the final one. The more steps we introduce the longer will be the time to recreate a gesture and thus the *swipe* will be slower. We want them to be as fast as possible but using not enough step can make the gestures fails. To find the right trade-off we calculate the distance between the starting-point and the ending-point and we divide it into steps depending on the length with the goal to create every time steps of the same length.

4.3 Logs Structure

When the injected code described in Section 3.3 intercepts the execution of functions that retrieve data from the device, or the access to mobile-specific properties, has to write a *report* on the **logfile**. Those lines in the *logfile* must be identified by a specific tag to make the elaboration of the results, at the end of the *scraping* process, easier and more efficient. Furthermore, we want them to contain all needed information to facilitate post-processing as well.

The logs are printed using the `console.log()` function provided by JavaScript. In Firefox for Android, all the logs generated by that function are saved in the Android OS system log with the *GeckoConsole* tag. Logs from our system will be categorized with the other ones generated by the browser.

The pieces of information we want to save in the log are stored in a **CSV** file, with fields separated by a *semicolon*. They are:

- **Timestamp:** it can be useful to understand if data is accessed at loading time or after navigation in the website. It is automatically registered by `adb logcat` command adding `-v time` option;
- **Type of access:** we want to differentiate the data accessed through methods and the one accessed through properties, plus as it will be better explained in Section 4.5.2 we want to use a specific *tag* for calls requesting explicit permissions by the user. The used tags are:
  - MWEBAPILOG for mobile-specific methods;
  - PROPMWEBAPILOG for mobile-specific properties;
  - MWEBAPILOGP for permission requesting methods.

  We choose the *tags* to have a common *root* (MWEBAPILOG), to have the possibility to choose how fine grained the research can be.
- **Identifier of the function called or property accessed:** we used the Object.property notation to indicate the name of the function or of the properties accessed by the website;
- **Domain of website:** is the current domain of the website that requests the data, it is retrieved through the location.href property of the window object;
- **URL of the source:** the source-file containing the JavaScript code used to gather information from the device. It is obtained using the `getScript()` function described in Section 3.3.3;
- **User Agent:** user agent is important to register that the interception happened from a mobile device. It indicates the used Android OS and Firefox version.

Summing up, the structure of the log is the following:

```
<timestamp> GeckoConsole: <type_tag> <domain> <source_file_url> <user_agent>
```

and this is an example of the output we expect in the *logfile*:

```
07-11 16:13:54.864 I/GeckoConsole(6872):
  MWEBAPILOGP;getCurrentPosition;us.justdial.com;
  https://us.justdial.com/mnbldr/d=&b=&g=hmpgjs&ver_=n4.50;
  Mozilla/5.0 (Android 7.1.2; Mobile; rv:59.0) Gecko/59.0 Firefox/59.0
```

Algorithm 1 FFAutomator core

```
procedure FFAutomator core
    do
        $current\_url \leftarrow$ read next line of the file containing websites
        Create new $logfile$
        Redirect the smartphone log output to $logfile$
        Start Firefox activity on the smartphone with $current\_url$
        $current\_time \leftarrow$ get current time
        $timeout\_time \leftarrow current\_time + timeout$
        do
            Simulate a user gesture
            $current\_time \leftarrow$ get current time
        while $current\_time \leq timeout\_time$
        Close $logfile$
        Download files containing functions
        Close tabs and clean the cache and the cookies
    while End-of-file is reached
end procedure
```

4.4 The Program

FFAutomator is the program we wrote to handle automatic visit of websites to harvest information from them. We decided to use Python3 to develop the program because of the wide range of available libraries and modules and because of the flexibility in interaction with the *Linux Shell*. As described in Section 4.1, this is particularly important because we are mostly using *Linux shell* commands to interact with the devices. To do that, we use the subprocess module of Python. It contains functions used to execute external programs from the current procedure with the possibility to regulate their execution. It means that we can control the input/output redirection and if the call is *blocking* or not.

4.4.1 Structure

The *pseudo-code* contained in Algorithm 1 shows the core structure of the program with the main steps composing it. The program is based on two main loops:

- the external loop is used to read the **Alexa Top 1 Million list** file line by line. Each line of the file contains the name of the website preceded by a number representing its rank.
- the internal loop sends a touch action to the device at each iteration to simulate the navigation of a real user.

In the next sections, we analyze the different parts of the Algorithm 1 explaining the techniques used to complete each task.

4.4.2 Visiting websites

In Android, all *activities*, regardless of their status, are handled by the **Activity Manager**. The latter can be instrumented using the `am` command of the Android shell. Our goal is to create a new tab in Firefox browser that contains the domain of the website that we want to visit. To do that, we use the `start` command (belonging to the *Activity Manager API*) followed by those options:

- `-a android.intent.action.VIEW` to specify that the activity we are willing to start is a view;

- `-n org.mozilla.firefox/org.mozilla.gecko.BrowserApp` describes the component we want to be launched with the activity, preceded by the name of the package;
- `-d <domain>` used to define on which data the activity is acting. In our case, it represents the string that will be written in the *address bar.*

When launching a new *view* of the browser, a new **tab** is opened. Now, a huge amount of websites must be taken into account and each open tab requires allocation of resources by the phone. For those reasons, it is required that, after a website is browsed, all tabs are closed. We manage this, sending touch gestures to open the tab menu and the selecting *close all tabs* option from it.

4.4.3 Logs saving

To facilitate data processing, we decided to organize the logs in separate files, one for each website. As it is possible to see in Line 4 of Algorithm 1, when a new domain is read from the list, a new file is created by the program. The *logfile* is named after the domain of the website as it is read by the script. In this way, we can be sure that each *log* is unequivocal and there is no risk of overwriting them.

Once the file is created, to redirect the output of `adb shell logcat` to the *logfile*, the program creates a new process in background using the `subprocess.Popen` function that executes the *Linux shell* command `adb shell logcat`. Plus we set the created file as the *standard output* (`stdout`) of the process. The latter and the main program continue running in parallel so the logs are continuously written on file while the program interacts with the device. Once the website navigation is over, the process in the background is killed using the value returned by the function that created it and the file that contains the *log* is closed by the program.

To ensure that each file contains only the logs generated by the device while the website was browsed, we clean all the **logging buffers** on the device thanks to the command `adb logcat -b all -c`. An issue we faced, is that there may be a delay between the launch of the cleaning command and the effective flushing of buffers. This happens especially when the *adb server* on the computer has to manage several devices. To bypass this problem, after having launched the cleaning command, we read the first lines of the new logs and we compare them with the ones collected for the previous website. If they are the same it means that the *log buffers* were not properly cleaned, so we make the program wait 2 seconds and repeat the operation. In this way, we ensure that each file contains only logs from the associated website.

4.4.4 Number of gestures injected

The *internal loop* in the program, that can be seen from lines 9 to 12 of Algorithm 1 has the purpose of simulating navigation in the websites. Given the huge number of domains we have to explore, it is important to find the minimum amount of time that each domain should be visited, to catch all the functions we are interested in. In this way, we can maximize the number of websites visited without making the quality of the results worse.

There are two different ways to simulate web navigation:

- injecting a fixed number of gestures for each website, guaranteeing that each page can be visited with the same depth;
- deciding how much must be time spent on each page and then set a *timeout* for navigation. In this case, we don't know how many gestures are sent but we are sure that each page will be explored for the same amount of time.

To decide which is the best approach, we tested both methodologies on the same websites and then we analyzed the results. We run the program on the first 750 websites with the goal of navigating each domain for around 30 seconds, considering that there is also a *overhead* given by cleaning data related to the old session and by some checks that will be discussed later in Section 4.5.

We started with the first alternative, injecting 60 complete gestures in each website. Each page was visited for **47 seconds** (including the overhead described previously) and the websites using potentially dangerous calls we identified were 48. Then we introduced a timeout of 30 seconds for each website and we repeated the experiment on the same websites. The average time needed for each website was of 37 seconds while the number of actions injected decreased to an average of 40. After all, results remained the same with 49 websites identified. Results are summed up in Table 4.1.

Table 4.1. COMPARISON BETWEEN RESULTS ON FIRST 750 WEBSITES

	60 gestures injected	30 seconds timeout
Websites identified	48	49
Avg time on each website	47 s	37 s
Avg number of gestures injected	60	30

A slight difference in the results is expected and acceptable because the loading of a website can fail for several reasons that will be explained deeply in Section 4.5. For now is enough to say that the results found with the two different approaches can be considered similar. From them, we can infer that most of the websites retrieve the data we are interested in at loading time or, at least, after a short navigation. So, there is no point in injecting a big number of gestures and the best trade-off between the page exploration and time elapsed in visiting a website is using a 30 seconds *timeout*.

4.4.5 Cleaning the environment

After every website is visited, it is necessary to *clean* all the data related to it. It is important to avoid any interference between different websites. For example, it may happen that two domains redirect to the same website and so, the second time, it is retrieved by the **cache** of the browser. This is a behavior we want to avoid. Our goal is to bring the browser at same initial conditions for all websites in order not to make browsing on a page being influenced by any previous one.

To ensure that, we clean all *private data* saved by the browser after each navigation through the *settings →Clear Private Data* entry in the options menu. The position of the entries in the drop-down menu is fixed and does not change while using the program, so it possible to control it automatically by sending *taps* at the specific coordinates.

After testing websites we realized that the application got *stuck* after processing the first few websites. Even if the program continued working, no websites were loaded into the browser and obviously, no function was identified and registered. We do not know why this happens but we can assume it is caused by an increased usage of **RAM** in the device that causes the application to block. Plus, after long testing, it may happen that browser *crashes* while visiting a website, opening the dialog to report the error to Mozilla. To avoid these two faulty behaviors, we realized that it was necessary to close the *app* so that the process is killed and the RAM cleaned.

The first approach to solve this problem consisted of *killing* the process through the Android shell command **kill**. In this way, we solved problems regarding crashes but the application continued getting stuck after visiting a bunch of websites. Given that, we assumed that when we kill the process through the shell, the associated *activity* did not get removed by the list of recent applications in *multitasking menu*. To bring the device as much as possible at the initial state, after each website is visited and private data erased, we send to the smartphone the gestures related to opening the multitasking menu and clearing all the activities currently open (that usually consists in Firefox only).

As for private data cleaning, the position where *actions* must be executed is fixed and so it is possible to automatically do it after each navigation is over. The necessary actions consist in *tapping* the multitasking button on the *navigation bar* for the Nexus 5X or the menu *soft-button* for the OnePlus One and then tap on *Clear All* in the top-right corner of the screen.

4.4.6 Files saving

It is important to download the source files containing the functions we intercepted during navigation. In the *logfiles*, the lines related to a data access detection contain the URL of the source file as described in Section 3.3.3. Exploiting the `wget` command of Linux shell, it is possible to download any element from a website, given the associated URL.

Once the *logfile* is written, we open it and, through a manipulation using *Bash* language, we reach our goal *piping* the following commands:

- `cat` command to visualize the file;
- `grep` command to isolate the logs regarding the HTML functions we intercepted using the associated *TAG*;
- `cut` command with the the proper option to isolate the field containing the URL of the file from the complete string;
- `sort` and `uniq` commands to eliminate the duplicates;
- `wget` command with arguments passed thanks to `xargs` directive to download them.

This is the full list of commands used:

```
cat <name_of_logfile> |
grep WEBAPILOG |
cut -d ";" -f 4 |
sort |
uniq |
xargs wget --quiet -P <path_of_destintion_folder>
```

The file is saved with the same name it is stored in the web-server in a folder named after the website to easily search and identify them later.

The operation of downloading the sources can be long if the scripts are very big or if there are many of them. On the other hand, it is completely independent of the navigation of the websites because it involves only the files containing the logs stored in the computer. For this reason, to prevent it from blocking the main operations on Firefox that are time-sensitive, we use the `Popen` function from `subprocess` module to make it run in parallel without blocking the main thread.

4.5 Issues in automatic browsing

Once the *core* of the program was outlined we started some testing to figure out which are the issues in navigation that must be faced to avoid any misbehavior in the process. Our approach consisted of testing it and then fix the program every time a problem came up. In the following sections we describe those cases together with the techniques we used to mitigate them.

4.5.1 Redirection to external websites

When visiting a website, we used the simulation of touch gestures to explore the document but it happens that tapping on some links redirects to *third-parties* websites, belonging to different domains. This can *contaminate* the results creating **false positives** when the original website does not use any API of interest but can redirect to a website that uses them.

The ideal scenario to solve this issue consists in making the browser communicate with the *FFAutomator* to make it aware of a redirection so that it can take the proper countermeasures. As described in Section 4.1, the only way to make the browser exchange information with the program is exploiting the logs.

First of all, we need to allow redirection to a different website it it happens automatically when a website is loading. This case can be handled waiting a certain amount of seconds at loading time before starting the navigation. On the other hand, we want to detect redirection given by touch gestures. To do that we have to control the *domain* of the website every time an action is generated by *FFAutomator*. This is one of the reasons why we need **atomicity** when generating gestures on the phone and why a *fuzzer* does not suit our needs, as we explained in Section 4.2.2.

***RedirectionDetector* module** To communicate the current domain to the *FFAutomator* we developed an Xposed module called *RedirectionDetector* that is able to recognize when something is written in the address bar of the browser. We use the same approach followed to intercept declaration of *Sensor* objects in Section 3.5. We use `findAndHookMethod` function in Xposed framework to hook the `onDraw` method of the android.view.View class returning the *View* object. The module checks if the *Object* caught is an implementation of android.widget.TextView and if it is so, it must be the Firefox address bar. In this way, our code in the module is executed when the *URL* in the active tab of the browser changes.

As it is described in Section 4.4.1, the URL of the next website is *typed* in the bar all at once and same happens when a link is opened in a website. Given that there is no way for the program to type anything in the address bar, the code in the module is fired only in the two cases described above.

The code in the module then, checks whether the accessed string matches with a URL and prints it on the log after having eventually cleaned the protocol part (*http://* or *https://*) and *www.* at the beginning. To recognize the logs, we add `‘‘REDIRECTIONTO:’’` tag before the URL.

Handling redirection From the *FFAutomator* side, we implement a function called `checkCurrentDomain()` that extracts the last entry in the log that contains the *REDIRECTIONTO* tag exploiting the following bash series of commands:

```
cat <path_to_logfile> | grep -a ‘‘REDIRECTIONTO:’’ | tail -n 1
```

and redirecting the output to a variable using the `check_output` function from the *subprocess* module. The returned string is then processed and the domain is extracted. If there is nothing written in the address bar, an eventuality that may happen for example if the browser cannot resolve the *address* because of problems with internet connection, the function returns the string *"None"*.

The *current domain* is checked in three different moments during navigation of a website. They are:

- after the website loading is issued by the program, it waits for a certain time for the website to load (it will be discussed deeper in Section 4.5.5), and then, it retrieves the current domain. At this point, we assume that a possible *automatic redirection* has already been executed, and the address bar contains the *final* address. So, the value returned by `checkCurrentDomain()` function is the domain that will be stored by the program as a frame of reference to detect *redirections* later.

- before a gesture is injected into a website, to be sure that no action is done on a website that is not supposed to be loaded. In case the current domain is different by the *sample* one, a **back** action is executed on the phone to return to the correct page. Thanks to the *atomicity* of the injected events we are sure that it is not possible to have more than a redirection given by a tap on an external link in the page, for each iteration of the loop in the program.

- after the action is injected, to check if the gesture redirected navigation to another website. This is necessary because, in the possibility that an unwanted website is browsed, it is necessary to delete the part of the *logfile* that was generated while the wrong website was loaded. To do that, we create a *backup copy* of the *logfile* as soon as the control on the domain we make before injecting gestures is successful. In this way, we can be sure that the saved copy is consistent. If in the control done after an action is executed, we discover that the domain has changed, we inject **back** action in the device and we copy back the *backup logfile* instead of the new with a **rollback** operation. On the other hand, if no redirection happens, the backup file is deleted. All those operations are done exploiting `bash` commands through the `subprocess` module.

Redirection dead end In some websites, redirection brought to pages that prevent the user from going back, in this cases when the *back* action is sent to the device, instead of returning to the previous page, the current page is reloaded. To prevent navigation from remaining stuck in this dead end, the program sends the input to go back for two times, if after two attempts the *current domain* is still wrong, we reload the initial website as it is done at the beginning.

Handling old Domain What we noticed after the first tests, is that sometimes while Firefox is loading the new website, our module prints in the log the last domain associated with the previously visited website. It happened at the beginning before the website is loaded, and the current domain written in the address bar. It can be caused by the application that continues calling the *canvas* method with the old value.

To overcome this annoying problem, after the visit of a website is over, we store in a variable the last domain registered for that page. Then, we add in the `checkCurrentDomain()` function a control on the old domain. So, if the current domain discovered by the method is the same as the previous website, it is ignored.

4.5.2 Permissions acceptance

Among all the functions that can be called by a webpage exploiting the *HTML5 WebAPIs*, some of them, after being called, require an explicit approval of the user through a popup on the page, handled by the browser. Those functions are the following:

- Geolocation;
- Vibration;
- Media Capture (camera and microphone).

Accepting the usage of them is important to make the Operating System gather data from the device and find the correspondence between the system calls and the JavaScript functions.

In the *configuration* options accessible in Firefox by typing `about:config` in the address bar, it is possible to grant by default all those permissions. Those settings in the Android version of Firefox 59.0 are supported only for the Media Capture API, setting the

```
media.navigator.permission.disabled
```

to `true`.

For what concerns the Geolocation and the Vibration APIs, the program must grant permissions as soon as the dialog appears in the screen. Even in this case, the communication happens through the logs. After doing some tests, both with the dummy website and with real ones, we realized that the dialog always appears on the screen after the related function was called. Given that, we have already registered in the *logfile* when those functions are called, and consequently we have the information that should trigger the *FFAutomator* reaction, consisting in accepting the permission request.

Monitoring permission requests If the permission-related functions are called as soon as the page is loaded, we can assume that they will be fired during the time the program waits to load the page at the beginning. So, after the page is loaded, before injecting any action on the device, we control if there is any function that requires any permission. To identify them, we give a different *tag* to the strings in the log that represent those functions. The tag is **MWEBAPILOGP** and the command used to isolate those strings in the *logfile* is the bash command:

```
cat <path_to_logfile> | grep -a ‘‘MWEBAPILOGP’’ | tail -n 1
```

that returns the whole line of the file. This is then processed and the *type* of the function is extracted. Based on it, *FFAutomator* fires a function that sends touch gestures to accept the permission as the user would.

On the other hand, for the functions that are called after the injection of a gesture, the issue is that, if any gesture is injected on the screen while the dialog box is open, the latter will be closed and permission not accepted. Plus, being the actions randomly injected, there is the possibility that permissions are even **declined**. Thanks to the *atomicity* of navigation simulation, we can check if there is the need to *accept* permissions before injecting new gestures. In this case, any **race condition** between granting permission and reproducing user navigation should be avoided.

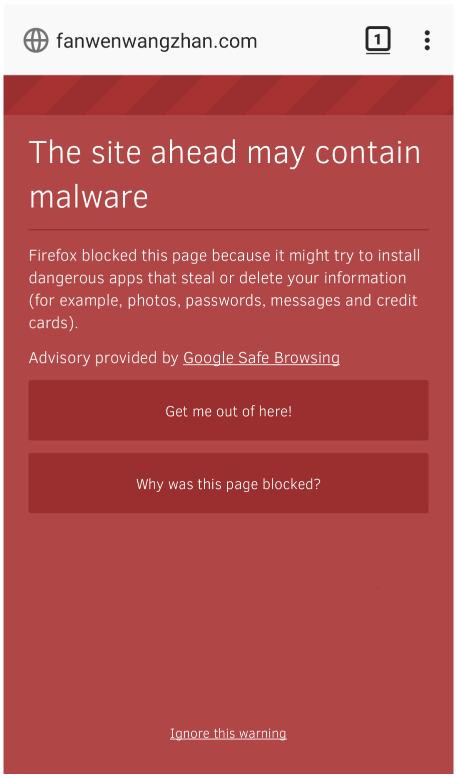
An issue we faced with this approach is that popups generated by the browser have always the same width, while the height can vary depending on how long the URL of the current website (that appears in the dialog) is. To mitigate this issue, we don't inject only one *tap* on the *"Allow"* button but, a series of *taps*, one below the other, each one far from the previous no more than the height of the button (that is always the same), to guarantee that the right area of the display, is pressed at least once.

To avoid that at next control on the *logfile*, an already considered string is taken into account and the consequent action fired, every time *FFAutomator* recognizes a function that needs explicit permissions, after accepting them, writes a *dummy* string in the same file. This string contains the tag *"DISCMWEBAPILOG"* and nothing else. At the next checking, if no other function is intercepted, the returned string will be the latter and the program will not start the *granting permissions* process.

Errors mitigation Considering the difficulties in synchronizing the *FFAutomator* and the device it is possible that some delay in loading the page or in writing the *logfile* makes this acceptance process fails. The only way to avoid this issue is enlarging the time waited before issuing any *action*. Even if with this approach the failings are drastically reduced, the navigation for each website will not be deep enough in the time slot dedicated for each page. For this reason, in the approach we followed, we *accept* a certain number of errors, but we log all of them, in order to reprocess those faulty websites later.

To check in which websites the program failed to grant permissions, we developed two different functions, one for *Geolocation* and one for *vibration* (*User Media* does not need it because the permissions are automatically granted by the browser), that check if those functions have been intercepted in the last *logfile* and, based on that, search for the associated *system calls*. The latter has been identified through the tests on our *dummy website*. After a website has been visited, the program checks if in the *logfile* there are the *permission protected* functions using the usual series of *bash commands* used for previous extractions of data from the files and exploiting the tag associated to those methods (*MWEBAPILOGP*). If they are found, the function subsequently checks, in a similar way, if the log contains also the Android system calls captured by the Xposed module described in Section 3.5. If no correspondence is found, the name of the website is written in a separate file so that it can be visited again later.

Limits in insecure connections Is worth mentioning that *Geolocation* function can be used only when the website supports **Secure Connections**. If the connection is *HTTP*, our system will detect the called function, but there will be no prompt and obviously, no Android system calls will be registered. In the log of the browser, it is registered with a specific message when some


fanwenwangzhan.com

The site ahead may contain malware

Firefox blocked this page because it might try to install dangerous apps that steal or delete your information (for example, photos, passwords, messages and credit cards).

Advisory provided by Google Safe Browsing

Figure 4.1. An example of how Firefox prevents users from visiting malicious websites

functions are called but results are not retrieved because of security reasons. This is handled by the program, searching in the *logfile*, if the string indicating the error in retrieving protected data (location in our case) is present. If it happens, no correspondence is expected, so those websites are not inserted in the file containing the faulty websites.

4.5.3 Malicious websites

What happened sometimes during navigation is that the *website*, even if it can be reached, it is blocked by Firefox because considered **deceptive** or containing **malwares**. Firefox browser helps users avoiding those pages integrating *version 4* of **Google Safe Browsing API** since the release of Firefox 56.

When a malicious website is visited, the screen represented in Fig. 4.1 appears to the user and prevents navigation. For this reason we do not have to navigate into them and they can be skipped when encountered. Google provides an API to communicate with this service and to know if a domain is in its list of dangerous ones. [6]

Exploiting that API, we wrote a function that sends an *HTTP POST* request to dedicated Google server, waits for its response and finally returns `True` if the domain is malicious or `False` if it is considered safe. Before starting visiting a website, the *FFAutomator* calls this function to check if it is worth to be visited. If not, its name is added in a proper file that, at the end of the testing operations, will keep track of all websites not visited because considered dangerous.

[6]More information about Google Safe Browsing API can be found at: `https://developers.google.com/safe-browsing/v4/`

4.5.4 Unreachable websites

A part of the websites failed to load when visited by the browser. This issue has several possible causes that can be grouped into two main categories:

- **Local network problems:** this category contains all issues related to our setup. It includes the misbehavior of the browser application on the device, a faulty internet connection in the private network and errors at the **proxy server** level that can fail in redirecting traffic. It may also happen that proxy server takes too much time to respond, inducing the browser in a timeout.
- **Server problems:** it includes errors that are independent of our configuration. It includes the impossibility in getting the web-server *IP* address because of *DNS* problems and the unreachable server, returning a *502 - BAD GATEWAY ERROR*.

The expected number of websites the browser fails to load is around the **10%** of the total websites visited. We can assume it considering the analogies between our study and the one from Englehardt et al. [71] that, scraping information from website contained in the same list we use, finds that percentage.

Client-side While it is impossible to be aware of server-side problems, for what concern our local network, we noticed that, after the *FFAutomator* launches the activity on the device, if the client cannot reach the server, it takes more than a minute for the browser to display the error page. Until that, there is no URL displayed in the address bar. For this reason, thanks to the *Xposed module* described in Section 4.5.1, the program can be aware if there was an error of that kind searching for the output from the module in the log. If no string associated to the redirection on the page is found it means that the website did not load yet. If, after an interval of time that will be discussed in Section 4.5.5, there is not yet any output in the log, we interrupt processing that website loading the next one and we record the name of the faulty domain in an error *logfile*.

Keeping track of those *failures* is important because, given that those errors are caused by temporary malfunctions, it is possible to visit them later when they will maybe be reachable and navigate them correctly.

4.5.5 Loading time

As we described in Section 4.4.4, we set a timeout before starting visiting the website and we stop when it is over. If the available *time slot* starts as soon as the activity is launched, there is the risk, especially for websites that contain plenty of contents, that most of the gestures are *ineffective* and useless because are injected while the website is still loading.

We can avoid this, waiting for *load* event to be fired by the website. To do that we inject, together with the script used to intercept functions, a *listener* to that event. When is caught, a string with the **"FULLYLOADED"** tag is written in the log. *FFAutomator* looks for this string in the *logfile* and starts browsing the page only when it finds it.

Faster loading To make loading of the websites faster, we block images loading in websites through Firefox settings. It is possible to do that going into the *about:config* menu and setting:

```
browser.image_blocking
```

to 0. A *placeholder* is set by the browser instead of all the images in the HTML document. *Images* are not necessary for the purpose of our study and there is no reason to think that blocking their download from the server can interfere with the regular behavior of the website.

Empirical approach Waiting for the complete loading of the website, in most of the cases, requires an amount of time that is not acceptable for the requirements we set about time spent on each web-page. What we did consists in waiting for at most 10 seconds that the website is loaded and after that, starting with events injection. Clearly, we check in the meantime if the *FULLYLOADED* tag is printed in the log and, in that case, we stop waiting and we start the navigation.

After testing 15000 websites, we measure the time interval between the launch of the activity and the registration of *FULLYLOADED* tag in the log. The average time is of 19 seconds. It is a long time compared to our expectations but it can be reasonable considering that all connections go through the proxy server. However, waiting 19 seconds for each website is not possible because would require a time slot too long for each page, but not waiting means wasting most of the time allocated for each website (as described in Section 4.4.4) while it is still loading. Plus, the *FULLYLOADED* tag is written when all objects in the document are loaded but it does not mean that the website cannot be visited in the meantime. There are cases when the page is not completely loaded but it can be still explored with touch gestures.

4.5.6 Unhandled corner cases

While we tried to address as many issues as possible, that we discovered after doing the first tests, there are some *corner cases* that we did not manage on purpose because they happened seldom and, avoiding them, would have required a lot of time more for each page making the approach no more suitable for our needs. Most of them are caused by the synchronization made through the logs, as said in Section 4.1, that does not guarantee a full control over the behavior of *FFAutomator*.

Two permission popups firing at the same time There is a very rare possibility when a website asks at the same time for *vibration* and *geolocation* access. The program will accept only the last one registered. We did not handle it for two reasons. First, this is a very remote possibility we never encountered in the first 15000 websites and second, the domain of the websites creating this issue is already registered in the proper *logfile* described in Section 4.5.2 and so can be post-processed later.

Redirections error We noticed that sometimes the process used to recover the old *logfile* when a redirection to a new website is detected, fails and we can find unwanted output in the file. Still, redirection process works and the homepage is restored. For this reason, we can recognize those files by processing the logs when the tests are over.

To evaluate if it was worth or not to better handle those cases we considered results from 15000 files. We developed a script that for each file extract the *domain* considered as the default one by *FFAutomator* when processing it. To do that we read the timestamp of each line of the file and we skip the first 10 seconds of output. Doing that we simulate the first 10 seconds the program waits to make the browser load the page. After that, we save all the other domains registered by *RedirectionDetector* module and then, every time we find a log related to a *mobile-related function*, we extract the domain originating it and, if it does not match with the original default, we control if it belongs to any of the other domains visited during navigation. Analyzing those results we did not find any unexpected result and everything was correct. Plus it would be easy to *clean* the logs removing all output logged between the unexpected domain and the restored correct one.

Chapter 5

Results

In this chapter we describe the results obtained by the study. We start with a *quantitative analysis* of the results, showing statistics about the number of websites we found using mobile-specific WebAPIs. Then, we critically discuss possible issues related to the number of websites exploiting those APIs. The results have been gathered from the first 100000 websites out of the **Alexa Top 1 Million** list.

5.1 Usage analysis

Starting from the files saved by the *FFAutomator* program, containing *raw* data, we categorize the usage of each API. So we will have as many files as many mobile-specific APIs we considered, containing the name of the domains exploiting it, together with the position in the **Alexa** list.

Table 5.1. NUMBER OF WEBSITES EXPLOITING THE APIs CONSIDERED IN THE STUDY ON THE MOST VISITED 200000 WEBSITES

WebAPIs	Number of websites exploiting it
Device orientation	2115
Geolocation	1605
Device motion	1296
Screen orientation change	652
Ambient light sensor	147
Proximity sensor	140
Vibration	57
Media capture	11
	6023

In Table 5.1 it is possible to see which of the APIs we considered are the most used among the most popular websites. Among the first 200000 entries of the list, 5101 websites exploit at least one of the considered APIs, while 766 of them use more than one at the same time. The most used are the ones related to acceleration and orientation sensors, plus the GPS data. The latter API is meaningful also on desktop computers. However we considered it as mobile-specific because only smartphones have an integrated GPS receiver while, when the latter is not present, it is retrieved not from the wireless geolocation and `mac` address identification.

Given that only Vibration, Media capture and Geolocation APIs, explicitly request permissions to the user, while this is not necessary for all the others, we analyzed how many websites retrieve information without making the user aware. Those websites are 3484 and issues related to this aspect will be discussed in Chapter 6.

5.1.1 Correspondence with Android calls

Starting from the list of files associated to websites exploiting mobile-specifc WebAPIs we look for traces of Android calls based on the WebAPI exploited. First of all we identified the Android call generated by the smartphones exploiting the *dummy websites* we talked about in Section 3.6. While for the APIs retrieving data using methods there is always a call accessing the Android API, for what concerns **sensors**, several functions hooked through the Xposed module described in Section 3.5 can be found.

The found hooked functions are the following:

```
getDefaultSensor(<Code and name of the sensor>)
getDefaultSensor(Game Rotation Vector)
getDefaultSensor(Rotation Vector)
getDefaultSensor(Tilt Detector)
orientationListener
```

The first function is called for all the sensor related APIs and the argument passed is the code and name of the sensor installed in the smartphone. They are different depending on the model of the phone. The other functions are called specific to the APIs related to the **orientation** and **acceleration** of the smartphone and the last one is specific for *orientation change*.

As it is possible to see in Table 5.2, most of the *calls* related to sensor have a corresponding system level call. On the other hand, the permission protected ones lack of the OS level counterpart. This is because of the failures in granting permissions to the websites requesting them.

Table 5.2. CORRESPONDENCE BETWEEN API CALLS AND ANDROID OS CALLS

WebAPIs	Websites containing corresponding Android calls	Percentage of consistent websites
Device orientation	2093	99%
Device motion	1288	99%
Screen orientation change	652	100%
Ambient light sensor	147	100%
Proximity sensor	139	99%
Permission-protected APIs	1126	65%

5.1.2 Distribution

The **distribution** of the websites using mobile-specific APIs among the first 100000 entries, is not *uniform*. The number of websites decreases, together with the *popularity* of the domains. This trend can be seen in the histogram in Fig. 5.1, where, in the first 5000 websites, the number of websites using the considered APIs is more than double the ones in the last chunk from 195001 to 200000.

A comparable evolution can be seen in the histogram in Fig. 5.2 showing only APIs that are not permission-protected. A possible reason for this behavior can be found in the *mobilization* of websites to guarantee a better experience to the users using smartphones. Given that, once the server knows the client is a smartphone, can load scripts to access data that would be useless to get with desktop configurations. Last, in the representation in Fig. 5.3, it is clear that after the first *slots*, the percentage of websites is almost uniform. This is probably caused by the exploitation of *Geolocation API* in desktop versions of websites, even if position is not retrieved by the GPS.

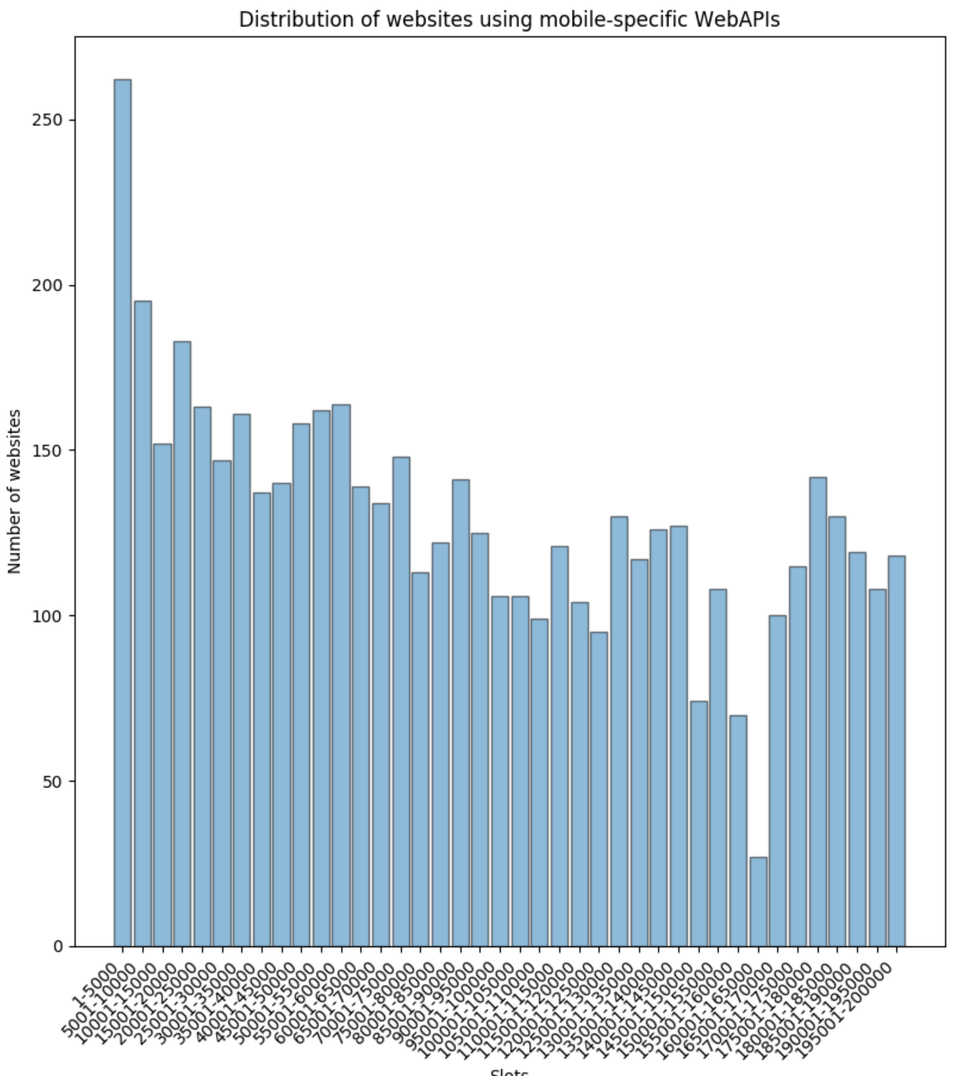


Figure 5.1. The histogram representing distribution of websites using mobile-specific WebAPIs

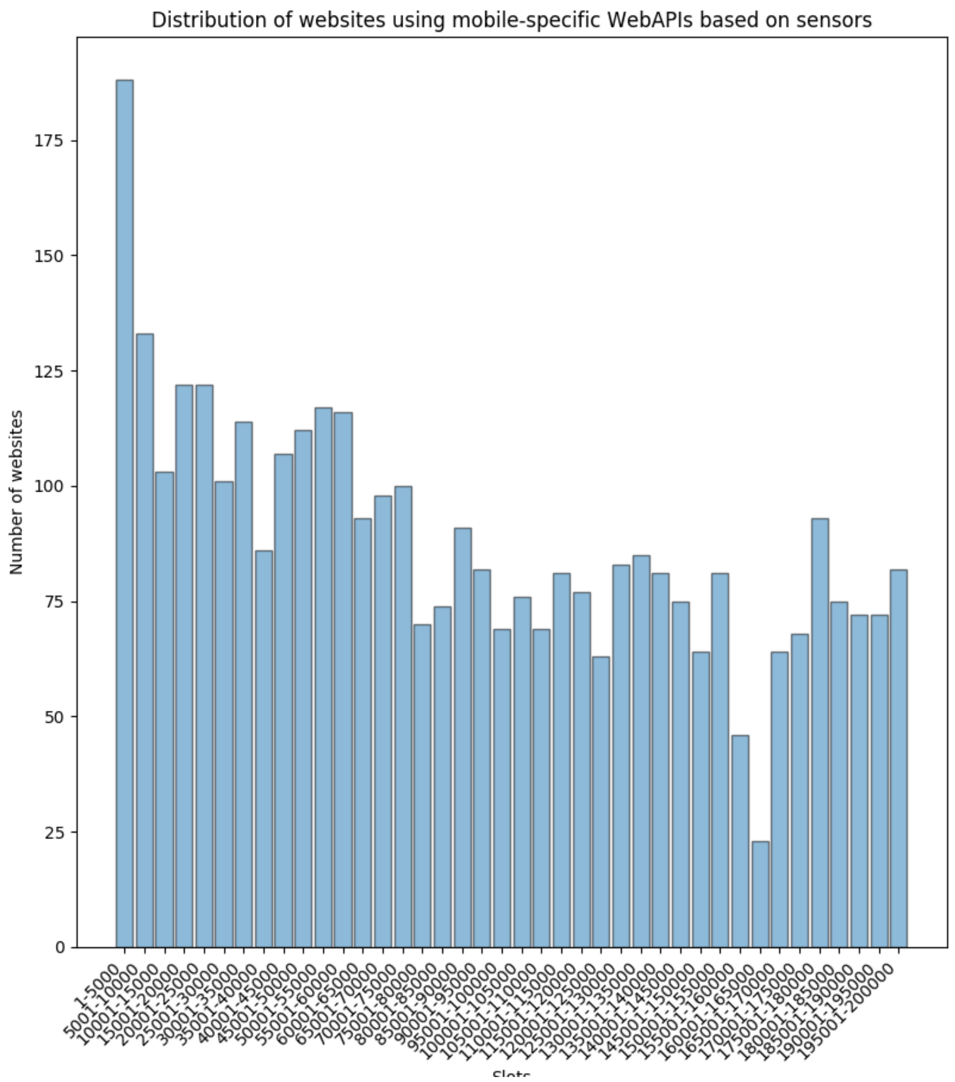


Figure 5.2. The histogram representing distribution of websites using mobile-specific WebAPIs based on sensors

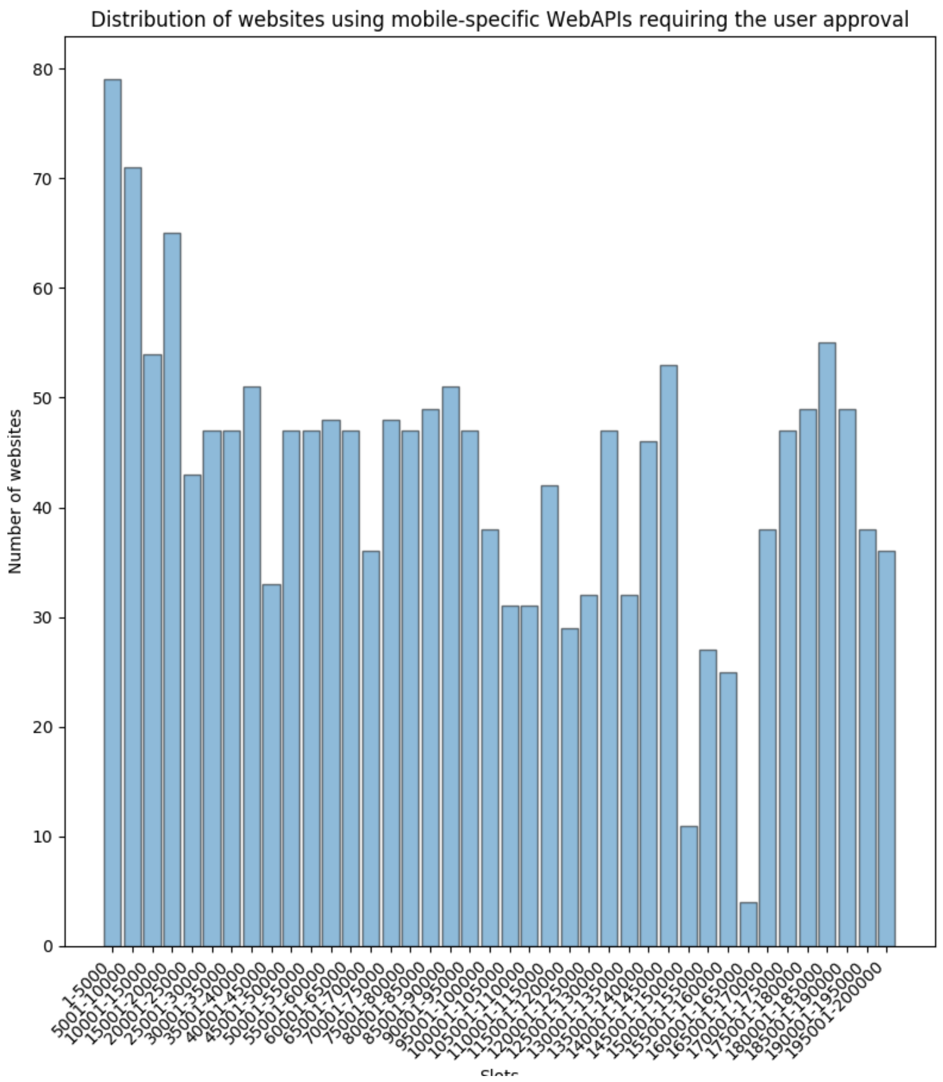


Figure 5.3. The histogram representing distribution of websites using mobile-specific WebAPIs requiring user approval

5.2 Sources of the calls

From the collected *logs* we deduced there are three different sources of scripts exploiting the mobile-specific WebAPIs. They are:

- **First-party website:** files are contained in the same domain of the website. We can assume that those libraries are loaded by the visited domain and do not rely on any external library. In this case, the *URL* of the current page and the one hosting the file are the same.
- **Third-party website:** files are loaded by an external sources but in the original domain. In this eventuality the *URL* is the same while the *path* of the document containing the JavaScript code belongs to a different domain.
- **iFrames:** files are loaded by an *iFrame*. The latter renders an external webpage inside the first one. In this case, injecting the code directly in the element, it is treated as a regular website. For this reason both the registered *URL* and the path of the source belong to the same external domain. To discern this case from a redirection of the visited domain to another one, and to make manage eventuality of not handled redirection, described in Section 4.5.1, we check if the *URL* is recorded by our *Xposed* module used to detect redirection. If it is so, we do not consider the log as originated by an *iFrame*.

5.2.1 iframes

As we explained in Section 3.6, we collect all the calls executed by every element of the website, including **iframes**. In the *logs* we save the domain of the source of the page gathering information. Saving the *URL* contained in the address bar in the *logfile*, we know exactly which is the current page and the domain of the source. Thanks to those pieces of information we save, we can figure out whether an API is exploited by the *DOM* or by an **iframe**.

We found that 946 websites out of 5101, contain **iframes** that use *WebAPIs* to access smartphone specific information. Among them we searched if domains contained in the *iframes* appeared in different websites. The two most frequent domains are related to online **media-players** and they sum up to 428 consisting in 45.2% of pages containing external websites collecting data.

5.2.2 Analysis of sources

Websites collecting information from a third-party source (including both *iFrames* and *external files*) we found that 40 external domains collect mobiles-specific information in 2514 websites, that is the 49% of all the websites we registered. Analyzing them, we found that they offer service for media-players and advertisement and they mainly collect information from **sensors** about the orientation and motion of the device. Considering only the sources that appear in more than 50 websites, we analyzed the category of the offered service and the type of data they collect, summarizing the results in Table 5.3. Analyzing the content of the latter, in the first column the domains hosting the files are listed. In the second one, there are the number of websites that execute the external sources while in the third is shown how many among those websites exploit iFrames to show that content. If it is so, we list the *URL* of the website in the frame in fourth column. Given that few third-party domains are used in more than one third of the websites, we analyzed the *classification* of them, to know the service they offer to the first-party website. We retrieve categories of the domains through *Cyren*[1], an online service to check that offer statistics about domains, to allow the users to check the safeness of a website. Last, in the sixth column, we list all the mobile-specific WebAPIs used by those domains.

It is possible to see how the services offered by third-party websites concern *advertisement*, *media streaming* and *shopping*. For what concerns the used WebAPIs, they reflect the numbers

[1]https://www.cyren.com/security-center/url-category-check

described in Table 5.1 except for *Geolocation API* that is exploited only by one of them while it is the second most popular in previous analysis. The usage of *orientation* and *orientation change* APIs can be associated to media streaming domains that need to render the content in portrait or landscape mode. On the other hand, most of them gather information about acceleration of the device through the *motion* API and most of them offer advertisement libraries. Only one of them read multiple sensors, getting data from proximity sensor and ambient light sensor in addition to acceleration and orientation.

It is important to underline how one the external sources found, *c.adsco.re*, falls into the category of **malware**. This website is not considered malicious by the **Google SafeBrowsing API** and so we decided to analyze the content of script retrieving the data. It came out that the same file retrieving information about *device motion* and *device orientation* is also responsible for accessing information that are well known to be exploited for device fingerprinting. It uses methods to create and manipulate *canvas* elements, plus it reads the properties in the next listing, that are well known to be used to fingerprint the device.

```
getTimezoneOffset
HTMLHtmlElement.clientHeight
HTMLHtmlElement.clientWidth
Navigator.cookieEnabled
Navigator.doNotTrack
Navigator.mimeTypes
Navigator.platform
Navigator.plugins
Navigator.userAgent
Screen.availHeight
Screen.availLeft
Screen.availTop
Screen.availWidth
Screen.colorDepth
Screen.height
Screen.left
Screen.top
Screen.width
Storage.getItem
Storage.setItem
Window.devicePixelRatio
Window.innerHeight
Window.innerWidth
```

Table 5.3. STATISTICS ON USAGE OF THIRD-PARTY SCRIPTS IN THE MOST POPULAR 200000 WEBSITES

Domain	Number of websites hosting it	Hosting through iFrames	iFrame domain	Classification	Exploited WebAPIs
fast.wistia.com	481	4	fast.wistia.com	computers and technology	orientation change
f.vimeocdn.com	312	312	player.vimeo.com	media streaming computers and technology	orientation
c.adsco.re	199	-	-	malware	motion orientation
cdn.admixer.net	176	-	-	advertisements and pop-ups	motion
g.alicdn.com	160	75	wanwang.aliyun.com m.aliyun.com	shopping general	location orientation
aeu.alicdn.com	139	91	mbest.aliexpress.com	shopping general	orientation
fast.wistia.net	130	116	fast.wistia.net	computers and technology	orientation change
static.yieldmo.com	96	-	-	advertisements and pop-ups	orientation
api.b2c.com	91	-	-	general	light motion orientation proximity
client.perimeterx.net	78	-	-	computers and technology	motion
dlswbr.baidu.com	72	70	pos.baidu.com	search engines and portals	motion
secure-ds.serving-sys.com	53	39	googleads.g.doubleclick.net tpc.googlesyndacation.com	advertisements and pop-ups	motion

Chapter 6

Extension

In this chapter we describe a Firefox **extension** we developed for the Android platform, to mitigate the possible dangers for users caused by mobile-specific data gathered by websites during navigation.

6.1 Why an extension

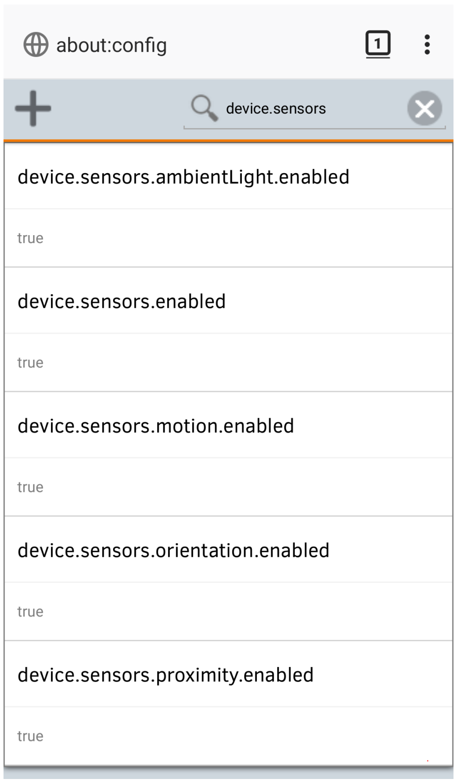
An **extension** is an easy and practical way for every user to implement functions on Firefox without requiring any additional knowledge of the browser. Firefox developers proved to care about the privacy of the users in many situations, addressing possible misuse of available WebAPI to harm user security. For example, they removed the support to the *Battery Status API* [38] from version 52 (2016) of the browser [40], after several studies [16, 39] proved it could be used to **fingerprint** the user through different sessions.

A growing concerning about user privacy led Mozilla to regulate the access to the hardware sensors we described in previous sections of our study in next version of the software. Until version 59 (the one we considered in our research), setting to `false` the `device.sensors.enabled` value in **about:config** tab, set to `true` by default, would prevent Firefox from communicating with all sensors in the device (i.e. proximity, ambient light, motion and orientation). In version 60 and 61, in the tab described before and that can be seen in Fig. 6.1 it is possible to enable/disable only specific sensors but all of them are set to `true` by default. For this reason, it is required an explicit action of the user to turn them off and it is not possible through regular settings but requires the access to the **about:config** tab. This procedure is not straightforward for less experienced users that would probably ignore this possibility.

With versions 62 and 63 (the most recent version that can be used at the moment) that are now respectively in *beta* and *nightly* stage, the options about the sensors remain the same with the only difference that *proximity* and *ambient-light* sensors are disabled by default.[1]

Another aspect that motivates the development of an extension is that rules set in the **about:config** tab are *global*. It means that they apply to all the tabs of the browser, without giving the user the possibility to customize the options for each website. With our extension users will have the chance to enable or disable the access to smartphone data, creating custom rules based on the domain. Plus our extension has the goal to detect which data is currently retrieved by the website to make the user aware of which pieces of information are accessed by the page.

[1]More information about release versions of Firefox can be found at: https://wiki.mozilla.org/Firefox/Roadmap/Updates


about:config

device.sensors

device.sensors.ambientLight.enabled
true

device.sensors.enabled
true

device.sensors.motion.enabled
true

device.sensors.orientation.enabled
true

device.sensors.proximity.enabled
true

Figure 6.1. The *about:config* tab with the options related to sensors

6.2 Architecture

The **MWAdetector** extension we developed is composed of two main parts, the **content script**, and the **dynamic HTML page** opened when the user taps on the plugin and that makes him set different options. The idea is to make the *content script* run before page loads, injecting the code to detect data retrieval and to block it depending on user settings. For what concerns intercepting HTML5 APIs, we use the same scripts exploiting in the website scraping and described in Section 3.3 with the proper modifications that will be discussed in Section 6.3.4. Most of the functions we used belong to **WebExtensions API** provided by Mozilla. [2]

What we want to detect The goal is to *detect* and possibly *block* all the APIs taken into account in the previous part of the study and described in Section 3.1. Summing up we want to intercept:

- **geolocation methods**
- **vibration methods**
- **media methods (camera and microphone)**
- **device orientation properties**
- **orientation change properties**
- **device motion properties**
- **ambient light properties**
- **proximity properties**

Clearly we need to distinguish the two ways data can be gathered, so if it is through a function or through direct access to events properties.

6.2.1 Content script

The content script in browser extensions run in the *context* of each web-page but in a separated *JavaScript environment*. For this reason, it is not possible to run hooking scripts in it because they cannot be aware of functions called in the *DOM* of the page. To bypass this limit, we use the content script to manipulate the *DOM* of the page and to inject the scripts in it, like it is described in Section 3.4.2, for the previous study. For the latter we exploited a Firefox version where defenses against code injection were disabled but in this case the extension must work on browsers as they are installed by the store. Plus, disabling those measures would expose the users to greater risks that are not balanced by benefits brought by the extensions. So, Firefox will prevent execution of in-line scripts in websites enforcing the CSP. We bypass those limitations, writing JavaScript code in extarnal files that will be packed together in the extension. Content script inject them as external sources and being the extension a *trusted* source, their execution will be allowed.

To make the injected code communicate with the *content script* whether a data is retrieved or not, we exploit the Firefox **CustomEvent API**.[3] The communication works in this way and can be seen in Fig. 6.2:

[2]Information about WebExtensions API can be found at: https://developer.mozilla.org/en-US/Add-ons/WebExtensions

[3]More information about custom events at: https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent

- **injected-code side:** whenever a call to a function or an access to a property is intercepted, a *custom event*, containing a tag related to what was intercepted, is fired. The method used is `dispatchEvent(<name of the event>, {detail:<intercepted_tag>})` from the `window` object.
- **content-script side:** in the content script, eight `boolean` variables are declared as `false`. Then, a *listener* to the custom event is declared and every time an event is caught, its content is checked and the corresponding variable is set to `true`. After the page is fully loaded, the content script contains information about which data is retrieved by the website.

6.2.2 Dynamic extension page

User interacts with the extension through a dynamic HTML page that is opened in a new tab when the user taps on the extension name in Firefox menu. It is developed as a regular webpage, exploiting *HTML*, *CSS* and *JavaScript*. When the extension page is opened, we want to display, first of all, which data is retrieved by the current page loaded. Unlikely the communication between *content script* and *injected code* we talked about in Section 6.2.1, in this case we cannot use events because those scripts run in different **contexts**. So, we exploit the **sendMessage** function included in the *runtime API* of *WebExtensions*. Exploiting the latter, it is possible to send *messages* from a script of the extension to another and we used it to create an *asynchronous communication* between the page and the content script.

Given that the *content script* contains all variables associated to the accessed data, we make it listen to a message and when it arrives, we send back a message to the extension page containing the *tags* related to the APIs we are intercepting and the associated value in a **JSON** format. When the *extension page* is opened by the user, the associated JavaScript code is executed and the first action it does consists in adding a listener to *messages* and sending a *dummy* message to the content script specifying the current active **tab** as the target. It is used only to notify the *content script* that the page is loading and is ready to receive a response. When the page script receives the information from the *content script*, it reads the JSON file and writes in the page the names of the API exploited by the website.

6.2.3 Local storage

In the *extension page* the user can also creates custom rules for each domain that are saved in local storage. The settings are modified by the user in the HTML page of the add-on while are read by the content script. The **content script** will read the stored values and, based on them, it will inject the code used to block a given API. Local storage can be seen as a persistent indirect communication channel between the two parts of the extension as can be seen in Fig. 6.2. The changing done by the users through the extension *HTML* interface will be immediately saved and will become effective at the following page loading. Local storage belongs to the browser but data is associated to the extension. It means that deleting the extension, all the saved data associated to it is deleted. On the other hand, memory remains until the extension is installed and killing Firefox does not affect it.

6.3 Programmer manual

In previous section we described the architecture of the extension and the approach we followed to provide the user the possibility to customize the behavior of the program. We explained also how the choice of the user become effective in the webpage and how the latter communicates back to the extension which mobile-specific API is exploited. In this section we will describe how each feature, previously described, has been implemented in the extension analyzing the components of the software.

To build the extension it is enough to create a `.xpi` archive containing all the files in *extension* folder that can be found in the archive containing the software developed in this study. As it will

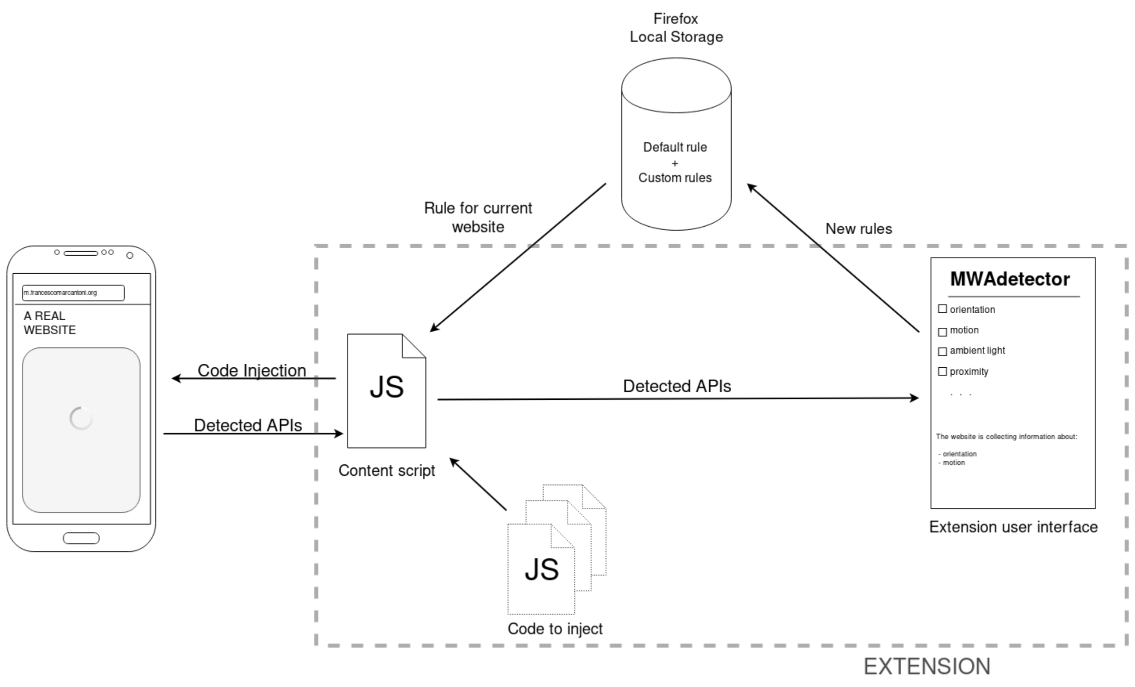


Figure 6.2. Diagram showing data flows in the extension

be described in Section 6.4, it is possible to directly build and move the extension to the device executing the *install.sh* script contained in the same folder. The content script (*injector.js*) and all the other scripts that are injected in the web-pages (*detector.js*, *propdetector.js*, *jamproximity.js*, *jammotion.js*, *jamvibration.js*, *jammedia.js*, *jamorientation.js*, *jamchange.js*, *jamposition.js* and *jamlight.js*) are contained in the same folder. Together with all need JavaScript files, it is possible to find the *popup* folder that contains all file requested by the popup interface to run. Being the extension page composed by two pages, it contains two couples of *HTML* and *JavaScript* files. They are *popup.html* and *popup.js* that creates the main page and then *rules.html* and *rules.js* for the page containing the table with the rules.

6.3.1 Permissions

The extension requires permission to run in the browser and be able to offer all the futures. Permission must be written in the manifest.json file and are requested to the user when the extension is installed. The portion of the manifest regarding permissions is the following:

```
"permissions": [
     "<all_urls>",
        "storage",
        "tabs"
  ]
```

- **all_urls** means that the extension can interact with all domains without restrictions;
- **storage** means that the extension have the access to the local storage of the browser. This is needed to save configurations regarding the *blocking rules*;
- **tabs** is used to have access to properties of the tab. In this extension we need this permission to have access to the current *URL* of the page and to send messages to the *content-script*.

```
{"custom-settings" : { "mozilla.org" : { "orientation" : false,
                                     "motion" : true,
                                     "light" : true,
                                     "proximity" : true,
                                     "change" : false,
                                     "position" : false,
                                     "media" : false,
                                     "vibration" : false } },
                   { "google.com" : { "orientation" : true,
                                    "motion" : true,
                                    "light" : true,
                                    "proximity" : true,
                                    "change" : false,
                                    "position" : false,
                                    "media" : false,
                                    "vibration" : false } },
 "default-settings" : { "orientation" : false,
                      "motion" : true,
                      "light" : true,
                      "proximity" : true,
                      "change" : false,
                      "position" : false,
                      "media" : false,
                      "vibration" : false }
}
```

Figure 6.3. JSON format of stored rules

6.3.2 Per website customization

The user settings are stored by the extension exploiting the Firefox **storage API**[4] included in *WebExtensions API*. Through this interface, an extension can save and retrieve data from the local storage of the browser. Information remains saved in memory through several sessions until the user deletes it.

Each custom rule is stored in JSON format, associating to each domain the *tags* of all the *APIs* we are willing to block, together with a `boolean` value that describes if the *API* must be blocked or not. All the loaded domains are contained in the `"custom-settings"` key. On the other hand, the *default* configuration is saved under the `"default-settings"` tag. An example of stored data can be seen in Fig. 6.3.

This structure makes data accessible as a **dictionary** having as *key* the saved domain and as value a similar structure with the name of the API as the key, and the boolean value containing information about the exploitation or not as value.

Data, formatted as shown in Fig. 6.3 can be stored in memory using the `set` function of *Storage API*. This is an example, where `stat` is the variable containing the set of *custom-rules*:

```
browser.storage.local.set({"custom-settings" : stat});
```

On the other hand, to get data from storage, the `get` method is used. The function returns a `Promise` object. To the latter we run the `then` method that executes the *callback function* passed

[4]Information about storage API can be found in: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage/local

```
  browser.storage.local.get(['default-settings']).then(function(data){
    stat = Object.values(data)[0];
    if (stat == null){
            // there is no data associated to 'custom-settings' key
    }
    else{
        // 'stat' contains the requested structure
      // and can be read here
    }
}, onError);
```

Figure 6.4. How data is read from storage

as first argument when the `promise` is returned. The callback function receives as argument the *data structure* and reads it. In case the *key* requested has no values associated, the `promise` will be `null`. An example showing how the custom rules are read can be seen in Fig. 6.4.

6.3.3 Content script (injector.js)

The **content script** of the extension, named *injector.js*, is a JavaScript code that is mainly used to inject the code in the website, collect the results coming from it and send the information to the extension page.

It is declared in the *manifest.json* of the as it follows:

```
"content_scripts": [
   {
     "run_at": "document_start",
     "match_about_blank": true,
     "all_frames": true,
     "matches": ["<all_urls>"],
     "js": ["injector.js"]
   }]
```

The content script has to inject code to intercept APIs usage, for this reason the code must be injected and executed as soon as possible. `document_start` option makes the extension execute the script every time a page starts loading. It is executed while the *DOM* is loading without waiting for the *document* to be ready. The script is loaded for all the domains, setting the `<all_urls>` pattern in `matches`. Plus, we set `true` to `all_frames` attribute, to extend the execution of the this script to all the frames of the webpage.

First script the script does is trying to retrieve information about the `default-settings` about APIs blocking from the browser storage using code in Fig. 6.4. The retrieved data is saved in 8 variables to be ready to use later if no `custom-settings` is found. If data retrieval fails, it means that it is the first execution of the extension, so all variables are set to `False` and the `default-settings` are stored in the browser for the first time. After that, the function injecting code is called (`activateJammer()`) and then the script declares listener to receive information about the detected APIs from the webpage, as described in Section 6.2.1.

The script listens to events properly fired by the injected script, adding proper listeners to the `window` object. Every time an event is caught, it contains the *tag* of the API that has been intercepted and a variable corresponding to it is updated. This is the code responsible of this task:

```
window.addEventListener("getChromeData", function(data) {
        if (data.detail == 'light')
              light = true;
        if (data.detail == 'proximity')
              proximity = true;
        if (data.detail == 'devicemotion')
              motion = true;
        if (data.detail == 'deviceorientation')
              orientation = true;
        if (data.detail == 'change')
              change = true;
        if (data.detail == 'position')
              position = true;
        if (data.detail == 'media')
              media = true;
        if (data.detail == 'vibration')
              vibration = true;
}, false);
```

The information related to the detected APIs, must be sent to the extension web-page providing *user interface* but the latter is active only when the user decide to open it. So it is necessary a synchronization between the two endpoints to avoid `injector.js` to send data when the pop-up is inactive. As we explained in Section 6.2.1, we make the script wait for a message from the *UI* that notifies the activation of the webpage and then the message containing the information will be sent. The function sending the message to the *UI* is the callback function of the listener used to be aware of *pop-up* activation. The code is the following:

```
browser.runtime.onMessage.addListener(msgfrompopup);

function msgfrompopup(msg) {
      browser.runtime.sendMessage({"light":light,
          "proximity":proximity,"motion":motion,"orientation":orientation,"change":change,
          "position":position, "media":media, "vibration":vibration});
}
```

The sent data is organized as a JSON file containing as key, the tag of the API and as value the boolean variable instantiated by the *content-script* and with the value obtained through the injected code.

activateJammer() This function, requiring no parameters, is used to inject in the website the JavaScript responsible for blocking data retrieval through the APIs we considered in this study. The function has to check if a *custom-rule* is set for the current domain and inject code based on it, otherwise the *default-rule* applies. The function tries to obtain all the rules associated to `custom-settings` in local storage, and saves the domain of the current browser tab, exploiting the `getDomain()` method that will be described later, in `dom` variable. If both the data structure retrieved by storage and `dom` are not `null`, the function tries to retrieve the rule for the `dom` among the found rules. If it exists, the code is injected following the rule, otherwise, the `injectDefault()` function is called. The latter reads the variables set at the beginning of script execution and inject the code based on them. On the contrary, if `dom` is `null`, on the other hand if `custom-settings` is not stored in memory, it means that no rule has never been applied by the user and so the `injectDefault()` function is called.

Fig. 6.5 shows the structure of the function, at the end another function is called (`injectDetectors()`) that injects the code to detect the calls, while the code is different, the way it is injected is the same and can be seen in Fig. 6.6. The file containing the code to inject (`jamorientation.js` in this example), must be in the same folder of `injector.js` when the extension is built. The function creates a new HTML element containing JavaScript code and adds as source the URL of the file obtained through `browser.extension.getURL()` from WebExtension API. The created element is insert in the `root` of the document that is the `head` of the *DOM* or the body if the first does not exist.

```
  function activateJammer(){
        browser.storage.local.get(["custom-settings"]).then(
               function(data){
                      stat = Object.values(data)[0];
                      dom = getDomain(location.href);
                      console.log(stat);
                      if (dom && stat!=null){
                             if (stat[dom] != null){
                                 if (stat[dom].orientation){ \\inject code
                                     blocking data about orientation
                                 }
                                 if (stat[dom].motion){ \\inject code
                                     blocking data about motion
                                 }
                                 if (stat[dom].light){ \\inject code blocking
                                     data about ambient light
                                 }
                                 if (stat[dom].proximity){ \\inject code
                                     blocking data about proximity
                                 }
                                 if (stat[dom].change){ \\inject code
                                     blocking data about orientation change
                                 }
                                 if (stat[dom].position){ \\inject code
                                     blocking data about position
                                 }
                                 if (stat[dom].media){ \\inject code blocking
                                     data from camera and microphone
                                 }
                                 if (stat[dom].vibration){ \\inject code
                                     blocking access to vibration engine
                                 }
                             }
                             else {
                                    console.log("Custom rule not found");
                                    injectDefault();
                             }
                      }
                      else {
                             if (dom==null){
                                    console.log("CURRENT DOMAIN NOT
                                        RECOGNIZED");
                             }
                             else{
                                    console.log("Applying default settings ")
                                    injectDefault();
                             }
                      }
               },onError);
  injectDetectors();
}
```

Figure 6.5. Structure of `activateJammer()` function

```
var jammer = document.createElement('script');
jammer.type = 'text/javascript';
jammer.async = false;
var jammerURL = browser.extension.getURL("jamorientation.js");
jammer.src = jammerURL;
root.insertBefore(jammer, root.firstChild);
```

Figure 6.6. Code used to inject code from the content script

getDomain(url) This function is used to retrieve the domain of the page opened in the current *active* tab. The *URL* is not enough because is not general enough, while we want a rule to apply for all subdomains and pages of the root website. The code is shown in Fig. 6.7 The function receives as argument the *URL* of the page extracted with the **Tabs API**. First step consists in removing the part related to the protocol (``http://'' or ``https://'') and then is cut all the portion of the *URL* after the symbols ``?'' and ``:'', in the case they are present. With this last operation we get rid of possible *HTTP queries* and *ports*. It is now tested if the *URL* is an explicit IP address, matching it with a regular expression. If the match fails, the remaining portion of the string is split based on the ``.'' and the last two field (the **domain** and **TLD** (top-level domain)) will compose the result. In the eventuality that the TLDs are composed by two fields separated by a dot (e.g. *co.uk*), we add another field to the result. Last we control if the obtained string contains *forbidden* characters. If the function is not able to build a correct **domain** returns `null`.

6.3.4 Injected code

Detecting

For what concerns the code exploited to detect access to data, we mostly reused the scripts used in Section 3.3, adapting parts of them to the new usage. The most important change is that now, instead of printing a string in the log, the code sends a *custom event* to the *content script* as described in Section 6.2.1. As opposed to the first part of the research involving scraping information from websites, here we cannot introduce *fake* but compliant data as we did in Section 3.3 because the user experience must remain untouched by the extensions. For this reason, the approach consisting in overwriting the `getters` of the properties of the *events* cannot be followed. On the other hand here we do not have time limits and we can wait for the website to be fully loaded, assuming that the user will wait for the page to be completely rendered before accessing the *extension*. It is possible to see in Fig. 6.8 the example of the interception of listeners instantiated through the assignment to the related DOM property. So, the injected code executes a method that checks if the properties that can be associated with a *listener* function are set when the `window.onload` event is fired by the DOM. A deeper analysis of this approach can be found in Section 3.3.

Blocking

To prevent websites from accessing data, according to user settings, we follow two different strategies according to the way information is accessed:

- **events:** in this case we follow the approach used in the first part of the study to detect access to properties. So, we substitute the `getters` for the properties of the events with *dummy* functions. In this way, after the event is caught by the page listener, the accessed properties return `null`. Code in Fig. 6.10 shows how properties related to the events fired by *sensors* are substituted with *null*.

```
  function getDomain(url){
        var host;
        var dom;
        var tmp;
        try{
               if (url.indexOf("://") > -1)
                      host = url.split('/')[2];
               else
                      host = url.split('/')[0];
               host = host.split(':')[0];
               host = host.split('?')[0];

               //support for ip addresses (e.g. local web servers)
               if (/^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
          (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
          (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.
          (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/.test(host))
                      return host;
               //we wanna block whole domain so we extract last 2 fields
                   divide by '.'
               tmp = host.split('.');
               if (tmp.length < 2)
                      throw "Domain too short";
               if (tmp.length > 2){
                      dom = tmp[tmp.length-2]+'.'+tmp[tmp.length-1];
                      if (tmp[tmp.length-2].length == 2 &&
                          tmp[tmp.length-1].length == 2 )
                             dom = tmp[tmp.length-3]+'.'+dom;
               }
               else
                      dom = host;
               if (/^[a-zA-Z0-9-.]+$/.test(dom))
                      return dom;
               else {
                      throw "Domain contains forbidden characters"
                      return null;
               }
        }
        catch(error){
               console.log("ERROR IN CHECKING DOMAIN CORRECTNESS:" +
                   error.toString());
               return null;
        }
}
```

Figure 6.7. Code of the `getDomain()` function

- **methods:** we substitute the methods used to retrieve data with *dummy* functions performing no action. It is possible just substituting the method property of the related *Object* with an empty one. The code is similar to the one used for event properties and shown in Fig. 6.10 with the only difference that now the whole method retrieving data is substituted with a blank function.

```
function checkifset (obj, prop){ if (obj[prop] != null){
  if (prop=='ondevicemotion') window.dispatchEvent(new
      CustomEvent('getChromeData', {detail: 'devicemotion'}));
  if (prop=='ondevicelight') window.dispatchEvent(new
      CustomEvent('getChromeData', {detail: 'light'}));
  if (prop=='ondeviceorientation') window.dispatchEvent(new
      CustomEvent('getChromeData', {detail: 'deviceorientation'}));
  if (prop=='ondeviceorientationabsolute') window.dispatchEvent(new
      CustomEvent('getChromeData', {detail: 'deviceorientation'}));
  if (prop=='ondeviceproximity') window.dispatchEvent(new
      CustomEvent('getChromeData', {detail: 'proximity'}));
  if (prop=='onuserproximity') window.dispatchEvent(new
      CustomEvent('getChromeData', {detail: 'proximity'}));
  }
}

window.onload = function() {
  checkifset(window,'ondevicemotion');
  checkifset(window,'ondevicelight');
  checkifset(window,'ondeviceorientation');
  checkifset(window,'ondeviceorientationabsolute');
  checkifset(window,'ondeviceproximity');
  checkifset(window,'onuserproximity');
}
```

Figure 6.8. Functions detecting listeners instantiated assigning a function to the `window` property.

```
hook(window,"addEventListener",function() {
 var array =
     ['deviceproximity','userproximity','devicelight','deviceorientation',
     'deviceorientationabsolute', 'devicemotion'];
  if (array.indexOf(arguments[0]) >= 0 ){
    if (arguments[0] == 'deviceorientation' || arguments[0] ==
        'deviceorientationabsolute')
      { window.dispatchEvent(new CustomEvent("getChromeData", {detail:
          "deviceorientation"})); }
    if (arguments[0] == 'devicemotion'){ window.dispatchEvent(new
        CustomEvent("getChromeData", {detail: "devicemotion"})); }
    if (arguments[0] == 'devicelight'){ window.dispatchEvent(new
        CustomEvent("getChromeData", {detail: "light"})); }
    if (arguments[0] == 'deviceproximity' || arguments[0] ==
        'userproximity' )
      { window.dispatchEvent(new CustomEvent("getChromeData", {detail:
          "proximity"})); } } });
```

Figure 6.9. Function used to intercept methods called by the webpage

6.3.5 Extension pop-up

The extension **pop-up** is developed as a regular webpage. It is built using *HTML5*, *CSS* and *JavaScript*. When it is opened there is an *homepage* and from this other sub-pages can be visited.

```
Object.defineProperty(DeviceOrientationEvent.prototype, 'absolute', {get:
    function(){} });
Object.defineProperty(DeviceOrientationEvent.prototype, 'alpha', {get:
    function(){} });
Object.defineProperty(DeviceOrientationEvent.prototype, 'beta', {get:
    function(){} });
Object.defineProperty(DeviceOrientationEvent.prototype, 'gamma', {get:
    function(){} });
```

Figure 6.10. Injected code used to substitute the getters to DeviceOrientationEvent properties with blank functions returning `null` value

In our extension, the *pop-up* consists in two pages. The first shows the user the results of analysis on the current webpage, displaying the detected APIs, and allow him to modify the default settings and create new rules for what concerns blocking them. The second page, that can be reached from the first one, contains a table with the currently set rules and allows the user to delete them. This section will focus on the scripts executed and not on the HTML interface that will be described in Section 6.4.

Main page (popup.js)

This script is executed when the popup page is loaded, that is when the *DOMContentLoaded* event is caught. At this point, the script updates the information about the *blocked calls* on the UI calling, the `updateBlockedCalls()` function. Then, it adds listeners to events fired by HTML page that are controlled by the user. They regard:

- adding a new custom rule or modifying an existing one (`customsubmission()`);
- modifying the default rule (`defaultsubmission()`);
- retrieve the domain from the current active tab (`getCurrentDomain()`);

Those functions are described in next paragraphs.

After having set those listeners, whose callback function will be discussed later, the page sends to the *content script* an *empty* message to notice it that the popup has been loaded, as described in Section 6.2.2, together with the message sending, the URL from the active tab is retrieved, elaborated through `getDomain()` function and stored in a global variable (named `domain`). Before starting the communication with the *content-script*, the page sets a listener to catch the response. When the latter is received a callback function is executed. This function parses the message, extracting the tags of the APIs that are exploited by the active tab and saving the information in boolean global variables, one for each API. The data are used by the function to write in the HTML page of the pop-up the name of the exploited APIs. After that, the `default-settings` are retrieved by the storage and the part of the page regarding the default rule is updated. The code is shown in Fig. 6.11. Information are displayed to the user writing it in a *paragraph* element for what concerns the detected APIs and *flagging* the checkbox in the menu used to change rules.

updateBlockedCalls() This function is used to update the list of the APIs that are blocked for the current website. They correspond to the *default rule* or to the *custom* one whet(her it is set. If the custom rules exists, also the checkboxs in the corresponding menu will be flagged. The domain is retrieved through the `getDomain(url)` function that is the same used in injector.js and described in Section 6.3.3. Using the domain, the value corresponding to `custom-settings` is retrieved from storage. From the retrieved value containing all custom rules, the one corresponding to the active domain is searched. If it exists, the contained values are read and the corresponing

```
function response(res){
        var changed = false;
        if (res.change || res.orientation || res.motion || res.light ||
            res.proximity || res.position || res.media || res.vibration){

                if (res.change){ ch = true;}
                ...
                if (res.vibration){ vi=true;}

        }
        if (or || mo || ch || pr || li){

                document.getElementById("callsdetectedtitle").innerHTML =
                    "<h5> Current website is collecting information
                    about:</h5>";
                par = "<p class=\"callsused\">";
                if (or)
                        par = par + "DEVICE ORIENTATION<br>";
        ...
                if (vi)
                        par = par + "VIBRATION<br>";

                document.getElementById("callsdetected").innerHTML =
                    par+"</p>";
        }
        else {
                document.getElementById("callsdetectedtitle").innerHTML =
                    "<h5> Current website isn't using any dangerous call</h5>";
        }

stat = browser.storage.local.get(["default-settings"]);
stat.then(readStatus, onError);
}

function readStatus(data){
        stat = Object.values(data)[0];

        if (stat.orientation)
                document.getElementById('ord').checked = true;
  ...
        if (stat.vibration)
                document.getElementById("vid").checked = true;
}
```

Figure 6.11. Functions used to update the detected calls and the default rule

names are written in the HTML page. If the domain does not exist in the set of custom rules or if `custom-settings` key does not exist (because no custom rule have been set), `default-settings` is retrieved and same actions take place.

customsubmission() and defaultsubmission() The functions are called when the related buttons are *tapped* in the *HTML* page. Functions read information from the checkbox menus and store the results in the browser *local storage.* These functions are wrappers respectively for `setCustom()` and `setDefault()` that actually create the JSON data structure and save them. The

```
function customsubmission(){
        dom = getDomain(document.getElementById("custdom").value);
        if (dom == null)
                document.getElementById('domainresponse').innerHTML = "Domain
                    not valid, please insert a correct one<br>";
        else{
                stat = browser.storage.local.get(["custom-settings"]);
                stat.then(setCustom, onError);
                console.log(JSON.stringify(stat));
                document.getElementById('domainresponse').innerHTML = "Added
                    domain: <span class='domainname'>"+dom+"</span><br>";
                }
}


function setCustom(data){
        dom = getDomain(document.getElementById('custdom').value);
        stat = Object.values(data)[0];
        if (stat == null){
                stat = {}
                console.log('no custom settings, adding: '+dom);
                stat[dom] = {
                        "orientation": document.getElementById("or").checked,
                        ...
                        "vibration": document.getElementById("vi").checked
                       }
                browser.storage.local.set({"custom-settings" : stat});
                console.log("ADDED DOM, EMPTY DICT");
        return;
        }
        else{
                console.log(dom);
                console.log(JSON.stringify(stat));
                stat[dom] = {
                        "orientation": document.getElementById("or").checked,
                        ...
                        "vibration": document.getElementById("vi").checked
                       }
                browser.storage.local.set({"custom-settings" : stat});
                console.log("ADDED DOM");
        return;
        }
}
```

Figure 6.12. The functions used to set custom rules. The functions for the default rule are the same without the controls described in the section.

differences between the two functions regards the need for `custom-settings` to control whether the domain is valid or not and if the `custom-settings` key exists in browser storage. In this case the structure is created, otherwise is modified adding the rule. The *domain* is typed in a form in the HTML page and retrieved by the script. In this case the user can add rules for domains without the need of visiting them. In Fig. 6.12 is shown the code used to set custom rules.

```
    browser.storage.local.get(["default-settings"]).then( function(data){
                stat = Object.values(data)[0];
                row = "<tr><td><b>Default</b></td>";
                if (stat.orientation)
                       row = row + "<td>Blocked</td>"
                else
                       row = row + "<td>Allowed</td>"
                ...
                if (stat.vibration)
                       row = row + "<td>Blocked</td>"
                else
                       row = row + "<td>Allowed</td>"

                row = row + "<td></td></tr>"
                var r = document.getElementById('rules').insertRow();
                r.innerHTML = row;
          },onError);
```

Figure 6.13. The code used to create a row of the table and to add it.

getCurrentDomain() This function assigns the value of the `domain` variable to the form where the user writes the domain they want to associate the custom rule. The user can execute this function, firing the event that reaches this script, when the domain they want to insert in the form is the one in the current active *tab*.

Rules page (rules.js)

The `rules.js` script handles the reviewing and the removal of the custom rules created by the user. The features handle the creation of a table containing the custom rules, the removal of a single rule, through a button close to the chosen line and last the removal of all the custom rules at once.

populateTable() This function is launched when the page is completely loaded, being inserted in a listener to `DOMContentLoaded`. The **table** HTML element already exists in the page and contains only the header. The function retrieves data from storage related to `default-settings`, creates the row of the table as it is shwon in Fig. 6.13 and adds it to the existing one. The same operation is then repated for all rules contained in the stored value corresponding to `custom-rules`. At the end of each custom rule, a button is added and the `deleteRule()` function is added to the `click` listener. Tapping on the button the custom rule is deleted and the corrsponding *row* too.

deleteRule(key) The function is used to delete the custom rule from the local storage and the row from the table. It receives as argument the *domain* that is the key to the rule contained in the storage, and is also the string shown in the table. To remove the rule from the storage, the `custom-settings` value is retrieved, the value associated to the URL is deleted and the structure is saved again. After that each line of the table is read and the one having the searched domain is deleted. The code of the function is shown in Fig. 6.14

cleanTable() This function is used to delete all the custom rules at once. The whole dictionary corresponding to `custom-settings` key in the browser storage is removed and then all lines in the table are deleted.

```
function deleteRule(key){
  var key = key.split('_')[1];
  console.log("DELETING "+key);
  browser.storage.local.get('custom-settings').then(function(data){
    stat = Object.values(data)[0];
    delete stat[key];
    browser.storage.local.set({'custom-settings' : stat});
  }, onError);
  var table = document.getElementById("rules");
  for (var i = 0, row; row = table.rows[i]; i++) {
    console.log(row.cells[0].innerHTML);
    if (row.cells[0].innerHTML.substring(3, row.cells[0].innerHTML.length-4)
        == key){
      table.deleteRow(i);
    }
  }
}
```

Figure 6.14. The code of `deleteRule(key)`

```
  zip -r detector@genso.com.xpi *
  adb -s $1 push detector@genso.com.xpi /sdcard
  rm detector@genso.com.xpi
```

Figure 6.15. Content of the script that creates the extension archive and moves it in the device

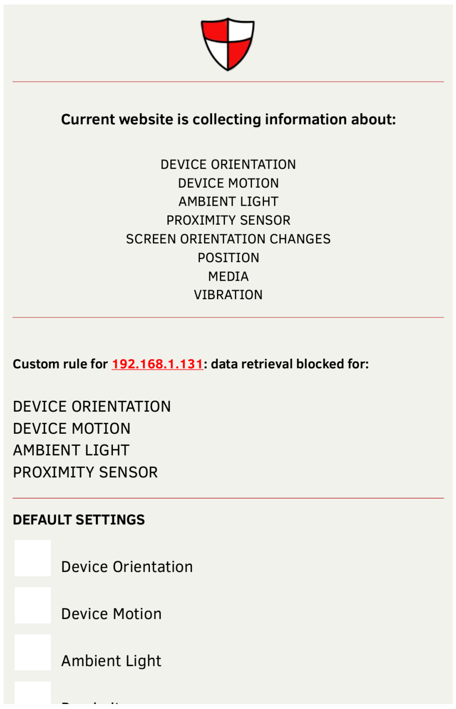
6.4 User Manual

6.4.1 Installation guide

The extensions is not available on the official add-ons store, but can be easily installed on any Android smartphone having **Firefox** browser installed.The extension has been tested in Firefox 59.0 and more recent versions, and it requires a computer running a Linux based OS with `adb` tool installed.

In the folder containing all needed files for the extension there is a *bash* script called **install.sh**. This script, which code can be seen in Fig. 6.15 creates the package of the extension and sends it the device that must have `adb` debugging enabled in developer settings (a guide is available at `https://developer.android.com/studio/command-line/adb#Enabling`). The script must be executed by command-line passing as only argument the *device identifier* using: `$ ./install.sh <serial_id>`. The script *zips* all the content in a `.xpi` archive, pushes it in the device memory and then deletes it.

Before installing the extension, it is necessary to enable in the browser configuration page, the installation of *untrusted* `xpi` files. To do that it is enough to type `about:config` on the address bar and set to `false` the entry named `xpinstall.signatures.required`. At this point, from Firefox browser on the smartphone it necessary to type `file:///sdcard` in the *address bar* and tap on the name of the extension (by default `detector@genso.com.xpi`) that is contained in the *install.sh* script. After that the extension is installed.



Current website is collecting information about:

DEVICE ORIENTATION
DEVICE MOTION
AMBIENT LIGHT
PROXIMITY SENSOR
SCREEN ORIENTATION CHANGES
POSITION
MEDIA
VIBRATION

Custom rule for 192.168.1.131: data retrieval blocked for:

DEVICE ORIENTATION
DEVICE MOTION
AMBIENT LIGHT
PROXIMITY SENSOR

DEFAULT SETTINGS

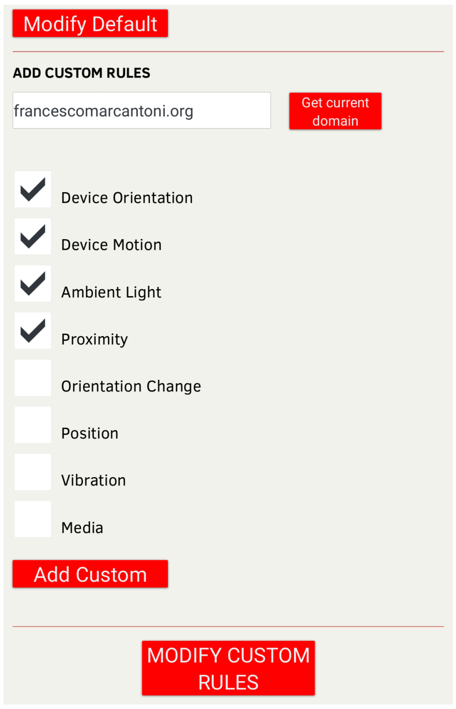
- [ ] Device Orientation
- [ ] Device Motion
- [ ] Ambient Light

Figure 6.16. The extension page containing the data accessed by the website and the blocked ones

6.4.2 User Interface

The interface of the extension, that can be seen in Fig. 6.16, displays on the top the eventual data that is accessed by the website in the current tab and then, below, shows which APIs are blocked for the current website. The user can modify the default rules that will be applied by default to all the *domains* or they can type a domain (or retrieve the current one that is visited through the proper button) and choose custom rules for it as it is shown in Fig. 6.17. In this way, the user can intuitively create *whitelists*, setting a predefined standard and then creating different rules for specific domains.

From the main page the user can switch to the **rules page** that can be seen in Fig. 6.18 where they can see the overview of the *custom* environments for all the domains and eventually remove the unwanted ones singularly or clear them all. The way custom rules are handled by the extension will be analyzed in Section 6.3.2.

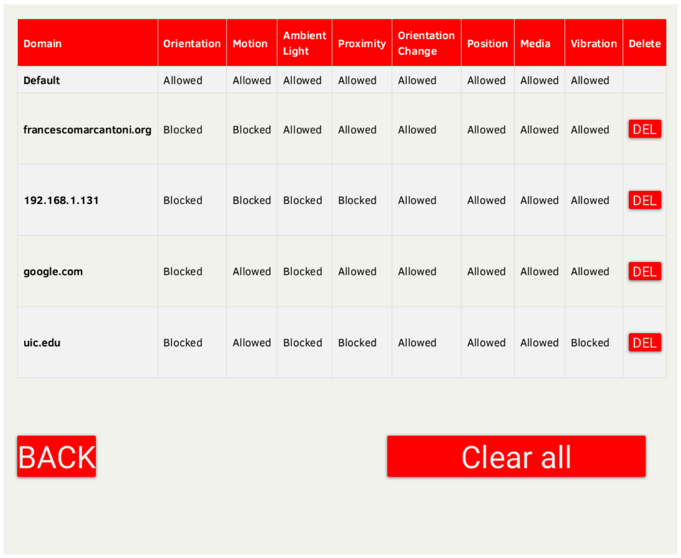


ADD CUSTOM RULES

francescomarcantoni.org

- [x] Device Orientation
- [x] Device Motion
- [x] Ambient Light
- [x] Proximity
- [ ] Orientation Change
- [ ] Position
- [ ] Vibration
- [ ] Media

Figure 6.17. The extension page providing customization for blocking rules



Domain	Orientation	Motion	Ambient Light	Proximity	Orientation Change	Position	Media	Vibration	Delete
Default	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	
francescomarcantoni.org	Blocked	Blocked	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	DEL
192.168.1.131	Blocked	Blocked	Blocked	Blocked	Allowed	Allowed	Allowed	Allowed	DEL
google.com	Blocked	Allowed	Blocked	Allowed	Allowed	Allowed	Allowed	Allowed	DEL
uic.edu	Blocked	Allowed	Blocked	Blocked	Allowed	Allowed	Allowed	Blocked	DEL

Figure 6.18. The extension page containing the custom rules set by the user

Chapter 7

Conclusion

The results from the study show how many among the most popular websites access data from smartphones without making the user aware of it and even if Firefox browser is trying to mitigate this issue, disabling a part of them, there is nowadays a risk for the user. Most of the scripts gathering information from the device are stored in third-party domains and are responsible for most of data retrieval. The remote web-servers hosting those script collect information from several web-pages, with the possibility to exploit characteristics of smartphones to track the user during navigation.

The usage of real smartphones in our approach, even if required a bigger effort caused by the introduction of complex systems as Android phones in the study, led to the creation of a faithful replica of a real navigation executed by a user. We can be sure, in this way of two things: first, that APIs detected with our approaches are executed also when a real user visit it; second, that the page is not aware of the simulation and can't limit the execution of some scripts. The simulation of touch events done through the *FFAutomator* script helps in simulating a real user browsing the web and contributed in improving the result, even if most of calls happen at loading time. On the other hand, it is impossible to visit portions of websites that require the *login* of the user to be accessed. For example we could not study the behavior of personal pages of the users that are present in many kind of websites, from social-networks to e-commerce.

Detection of APIs done through a proxy server makes the followed approach suitable to be applied also in other contexts. For example if the application used to browse the net changes or if even only the version is modified, no further adaptation is required and our approach can be easily replicated.

7.1 Future works

For what concerns the elaboration on results, it would be interesting to cross-reference the sources of the JavaScript files detected by our study with the ones extracted by other studies that detected websites gathering data that can be exploited for **fingerprinting**. Given the many well-documented proof-of-concepts about the usage of sensors data to fingerprint the habits of the user, it would be interesting to search if the found domains are known to collect also data to fingerprint navigation.

It would be interesting to see if the most popular **ad-blockers** and **privacy** enforcing extension block the external domains we found in our study. Plus, it would be worth to replicate our same study using one of the mobile-browser developed to preserve the privacy of the user (an example is Mozilla Firefox Focus[1]), to see if the issues created by mobile-specific WebAPIs are addressed.

[1] more information about Firefox Focus can be found at: `https://support.mozilla.org/en-US/kb/focus`

Our study focus only on Firefox browser for the reasons explained in Section 3.6.2, but there are also other very popular mobile-browser that are currently used. The most important is probably **Google Chrome**. The latter supports most of the APIs intercepted in the research but, in addition, has support to Generic Sensor API[2]. The latter provides an unified API that contains all functions to read data collected by all the sensors in the smartphone. Being this API not supported by Firefox, website using them were not detected. It would be interesting to be able to replicate our study on same websites but on different browsers to check how dangers for the users change, depending on the application used to navigate.

It would be interesting also to port **MWAdetector** extension to other mobile browsers. Now, Firefox is the online mobile browser that guarantees support for extensions but, given the increasing popularity of mobile browsing, it is not unlikely a support to them in the short period. In that case it would be useful to extend our software to a greater number of applications in order to reach as many people as possible.

[2]further information about Generic Sensor API can be found at: https://www.w3.org/TR/generic-sensor/

Appendix A

FFAutomator manuals

A.1 User manual

FFAutomator has been developed as a script, so it does not need to be installed. The software is composed by a main script and a separated module exploited to communicate with *Google SafeBrowsing API*. Together with a bash script named *scriptsdownloader.sh* that is responsible for downloading JavaScript sources. All the scripts need to be in the same directory. The software was developed in Python3 and was run on `Arch Linux` with kernel 4.16.8.

A.1.1 Prerequisites

- **Python3:** Python3 interpreter can be installed in any Linux distribution exploiting almost all package managers. In *Arch-based* distributions it can be installed using **pacman** through the command `pacman -S python3`. The version of the interpreter we used is Python3.6.5.
- **Android debug bridge (adb):** this tool is necessary to make the script communicate with the Android mobile device. It provided with the Android SDK, or it can be installed in *Arch-based* distributions with the `android-tools` package through the command `pacman -S android-tools`. The version we used is 1.0.39.
- **Firefox for Android:** our study focused on Firefox for Android v. 59.0 but the *FFAutomator* can instrument navigation in any Firefox version.
- **Android OS device:** we exploited Android version 7.1.2. While there is no requirement for Firefox version, our software may not work with other releases, mostly because the approach we used to simulate the user gestures on the screen may not work. It is also necessary that `root` privileges are acquired because *FFAutomator* needs them for some crucial operations. Plus, the software relies on logs produced by Xposed modules, so the framework must be installed, together with the three modules we provide together with the software.

  For what concerns the device, the software is developed to work with OnePlus One and LG Nexus 5X. The way we simulate gestures changes for each device, so they may not work on other models.

A.1.2 Input

The program is structured to be easily used in parallel with more than one mobile device, dividing the list containing the ranks of the domain in smaller ones. The domains to visit must be contained in a `csv` file, structured as the *Alexa list*. It means that each line contains the rank of the website (starting from 1) and the domain, separated by a comma (`<rank>,<domain>`). The websites are sorted in ascending order with no gaps.

To facilitate the visit of the websites we decided to make the execution of each script independent from the others. When multiple devices are exploited, the list must be divided in N smaller lists where N is the number of smartphones used. Each obtained *sublist* contains websites that are N positions far from the previous. It means that if we have 4 devices, we will have the following lists:

- **list 1** contains websites 1,5,9,13...
- **list 2** contains websites 2,6,10,14...
- **list 3** contains websites 3,7,11,15...
- **list 4** contains websites 4,8,12,16...

Using this division, if we stop all the scripts at the same time, the visited websites will form a complete *chunk* with no gaps inside.

The arguments that must be passed to the script in order are:

- **Serial number of the smartphone:** is the *alphanumeric* identifier of the Android device, it is used by `adb` to communicate with a smartphone when several of them are connected at the same time. This serial number can be seen through Android debug bridge with the command `adb devices`.
- **Number of devices used:** integer indicating the number of the devices used. It also indicates in how many *sublists*, the original list is divided into. This information is useful to obtain the line of the file we have to start at, depending on the number of devices used, the line of each file will contain a different website.
- **Rank of the starting website:** integer indicating the rank of the website that must be visited at the beginning, skipping all the previous ones. If the script is interrupted, it is possible to restart it exactly where it was stopped.
- **Model of the device:** this string indicates which device model is currently used. The supported models are the OnePlus One, that corresponds to `opo` and the LG Nexus 5X that is identified with `nex`.
- **Path to the file:** is the path of the file containing the list of websites. Each file must be organized as described before.

The program is thought to be launched from command-line in a Linux based OS. A usage example is:

```
$ python3 ffautomator.py abc123 4 9 opo file1.csv
```

the program is called to communicate with a OnePlus One with `abc123` as device serial. It is used with other three devices and the list of domains is contained in `file1.csv` that contains the domain with rank 9 that must be the first website to be visited.

A.1.3 Output

For each website, the program generates a *logfile* named after the visited domain. Those files are contained in a folder named after the `id` of the device. Each file contains all the logs generated by the device during navigation of a website. Furthermore, all JavaScript files containing code to access sensitive information of the device are stored into a folder named after the domain and stored in a directory named `jsfiles`.

The program stores also information about the execution of the program itself. It is saved in several files, each one related to a different issue. They are:

- **loadingfailslog:** contains the name of the websites that are not fully loaded when navigation starts and websites that fail to be loaded. First category is signaled with a `WARNING` string while the second one with the `ERROR` one.
- **failedgrantingpos** and **failedgrantingvib:** contain the names of the domain that ask respectively for the access to the GPS position and for the control on vibration engine but whose permissions were not granted by the program.
- **unsafewebsites:** contains the domain that are considered malicious by the Google SafeBrowsing API.
- **rollbacklog:** keeps track of any redirection that happened during navigation and the consequent rollback to the previous domain. Each log describes the two domains interested in the faulty redirection.

A.1.4 Google SafeBrowsing API

The python `sb` module we developed is used to communicate with *Google SafeBrowsing API*. The user requires to register an account to access that API (here is a guide to do that: `https://developers.google.com/safe-browsing/v4/get-started`).

After that the `sb.py` file must be modified, modifying the value of `apikey` variable with the key provided after registration and `clientId` field in JSON variable `body` with the created username. After that the module will send *lookup* requests automatically when called by the main script.

A.2 Programmer manual

The program consists in a main module, `ffautomator.py`, that provides the core functionalities of the software. That is, navigation through the websites including the simulation of user interactions and the handling of misbehaviors that can lead to poor quality results.

Being the program a script it does not require compilation and it is enough to execute it with Python3. In the archive it is contained in *automator* folder, together with the other scripts that will be described later and that are *sb.py* and *scriptsdownloader.sh*. After the execution of the program all the output files will be contained in the same folder too.

The script is composed by a main loop that is responsible for the automatic navigation of the websites on the smartphone, by an initial part when variables and file are set up and last by several functions that are used to control the flow of web-navigation and to simulate user interactions.

A.2.1 Environment setup

As a premise, it is important to point out that, given the large usage of interactions with the `adb` command in Linux shell, we used the `subprocess` module that can be imported in Python. It provides functions to execute *command-line* programs and script. We mainly use two functions of that module. They are `run` that interrupts the flow of the script, until the launched command returns and `Popen` that detaches the process without interrupting the current one. Plus the function `check_output` when it is necessary to read the output of the command. Documentation about the `subprocess` module can be found at `https://docs.python.org/3.6/library/subprocess.html`.

Arguments

Once launched, the script looks at arguments passed through command line, verifying their correctness and associating them to variables:

- **serial:** variable containing the serial string used to identify the device by `adb` and that is passed as first argument. It will be used in the script, every time an `adb` command is called.
- **firstsite:** variable containing the number corresponding to the line where the reading of the file with the list of domains must start. It is obtained thanks to the second and third arguments that correspond to the number of devices used and the *rank* of the website where navigation should begin.
- **model:** the model of the device used that is get from fourth argument. This variable is used every time different actions must be executed based on the smartphone model. The supported models are the OnePlus One and the LG Nexus 5X. Other variables are set at the beginning, they correspond to the resolution of the screen (**x** and **y**) and to the pixel width of the *navigation bar* (**nav**) that is exploited by some android devices.
- **path:** the path of the *csv* file containing the domains to visit.

Output files

While files containing the *logs* from the devices are created in the main loop for each website, the ones containing information about the correctness of the results and actions done by the software to control the navigation, are created at the beginning. In case the file already exists they are opened, in this way if the program is stopped and restarted later, the file will continue being written without loosing any data. This is the part of code that creates/opens files:

```
rollbacklog = open('rollbacklog', 'a+', 1) # logs of any redirection that required a
    rollback
loadingfailslog = open('loadingfailslog', 'a+', 1) # logs of incomplete or failing
    websites loading
unsafewebsites = open('unsafewebsites', 'a+', 1) # logs of websites considered
    malicious by SafeBrowsing API
failedgrantingpos = open('failedgrantingpos', 'a+', 1) # logs of websites accessing
    position but script failed granting it
failedgrantingvib = open('failedgrantingvib', 'a+', 1) # logs of websites accessing
    vibration but script failed granting it
```

Global variables

Some global variables have been used when it was necessary to share information among different functions. They are:

- **firstrowlog:** contains the string corresponding to the first row of the *logfile* created for each website. It is used to check whether the logs of the device are correctly flushed after every website. Before saving the logs, the first row is read and if it is the same to the one contained in this variable, the device *logs* are flushed again, otherwise it is stored in it. It is impossible that the two rows correspond because they contain the *timestamp* that will always be different. This variable is initialized to a series of symbols (that will never correspond to a real log) at the beginning of the program in order to consider the logs correctly cleaned at the first visit.
- **domainold:** contains the last domain registered in the log. It is used to avoid that the domain from the previous visited website *contaminates* the new logs. When the read domain corresponds to the old one, it is discarded.
- **initialdomain:** it is the domain read by the device as soon as the website is loaded. It is used as the basis of comparison to detect a possible redirection to a different domain.
- **currentdomain:** it is the domain currently visited by the browser. It is compared to `initialdomain` to spot a redirection.

- **pos** and **vib**: they are *boolean* variables that are set to false every time a new website is visited and turn to true when the access to the location or the the vibration engine is requested by the website. They are then used to check if permissions have been correctly granted by the script.

A.2.2 Main loop

The flow of the program is represented in Fig. A.1. The main steps represented in the algorithm are the following:

- **website launch:** csv file is parsed and read line by line through the proper Python3 module (`https://docs.python.org/3.6/library/csv.html`). It is read line by line and each line is parsed in an array to read different columns. In our case we have only 2 of them, the one at index 0 contains the rank of the domain while at index 1 we have the domain itself. Website can be launched in the mobile browser through an `adb` command to launch activities on Android. Then the command is executed using the `run` function. This is the code used to do that, in different parts of the script:

```
bashff = "adb -s " + serial + " shell am start -a android.intent.action.VIEW
    -n org.mozilla.firefox/org.mozilla.gecko.BrowserApp -d '"
...
command = bashff + row[1] + "'"
subprocess.run(command.split(" "))
```

- **log cleaning attempts:** the function that cleans the device *logcat* (`cleanLogcat()`) returns a value that indicates the success of the operation. If it fails, the function is repeated for at most 20 times, waiting two seconds between each attempt. Once the *log* is flushed, it is redirected to a file named after the current domain that will contain all the logs produced by the smartphone. The code is the following:

```
m = 0
result = False
while m < 20 and not result:
    result = cleanLogcat()
    if (result):
        print ("logcat cleaned")
    else:
        m = m + 1
        print ("ERROR cleaning logcat, retrying ... (attempt: " +str(m)+")"+"
            ["+row[1]+"]")
        time.sleep(2)
...
with open("logs"+serial+"/" + row[1] + ".txt","ab+") as out: # generates the
    logfile for the current website
    proc = subprocess.Popen(logcommand.split(" ") ,stdout=out)
```

- **website loaded:** once the website is launched, the scripts controls if it is completely loaded or not for at most 10 seconds. If after this amount of time the website is not completely loaded it is logged in the proper `loadingfailslog` file, as a *WARNING*. Otherwise the execution continues. After that the current domain is retrieved and saved as `initialdomain`. If the retrieval of the latter fails (the returned value is "None") navigation on the website is not simulated because it would be useless, and the event is reported in the same file used before but with the *ERROR* tag. This is the code handling it:

```
end_time = time.time() + 10 # set a 10seconds timer
            while time.time() < end_time and not checkLoading(): # wait for at
                most 10 seconds that the website fully loads
               pass

            if time.time() < end_time:
```

```
    print("OK - 100% LOADED")
else:
    print("NOT 100% LOADED")
    loadingfailslog.write(str(datetime.datetime.now()) +" WARNING:
        website is probably not 100% loaded;" + row[1] + "\n")
# --- READ INITIAL DOMAIN OF THE WEBSITE --- #

initialdomain = checkCurrentDomain()
currentdomain = initialdomain
```

- **navigation length control:** each website must be visited for a maximum amount of time. It is set before the loop starts and at each iteration the current time is read and, if the difference with the initial time saved at the beginning is greater than the chosen timeout, the loop is interrupted. Anyway, the iterations of the loop can be decided by the developer before execution, changing the range in the `for` cycle. In this way is possible to change the stop condition of navigation that is the number of injected events or total amount of time. At each iteration, a single gesture is sent through `sendEvent()` function and permissions are accepted with function `handlePermissions`.

- **rollback control:** in some parts during the simulation of the navigation, that can be seen in Fig. A.1, the scripts control whether a redirection happened in the browser to en external domain. If this happens, the script calls a function that send a *back event* to the smartphone in order to go to the correct website. Then, the lines saved in the logs and belonging to the wrong website must be deleted. To do that, every time we check the domain and we discover to be in the correct website, we copy the current *logfile* in a temporary file and we restore it if we discover that we are on a wrong domain. The correctness of current *URL* is controlled before sending a touch gesture and after it. Even if it can seem redundant, it was very useful to catch all possible wrong redirection. In the first control, if we spot that after the *rollback* the domain is still an unexpected one, it means that navigation reached a website that continuously reload even after a *back* event. In this case we reload the `initialdomain` to stop this *deadend*. It is important to notice that after reloading the website we force a *REDIRECTIONTO* tag with the correct initial website on the log to avoid any remaining log from the faulty redirection. The code, being longer than the previous ones reported, is in Fig. A.2. The code contains only the controls done before sending the events, the other one is the same but without the call to the functions to send the gestures (*sendEvent()*) and to accept permissions (*handlePermissions*).

- **download the scripts:** after navigation ends, all the scripts are downloaded. Depending on the number of functions found the downloading process can be long. For this reason we use an external bash script containing the command to download all the files and we run it with `subprocess.Popen` function that detach its execution from the main script. While downloading operations go on, the script continues its execution with the next website. This is the code:

  ```
  dl = "./scriptsdownloader.sh logs"+serial+"/"+row[1]+".txt" +" "+row[1]
  subprocess.Popen(dl.split(" "))
  ```

  and this is the content of the external bash script:

  ```
  cat $1 | grep WEBAPILOG | cut -d ";" -f 4 | sort | uniq | xargs wget --quiet
      -P "jsfiles/$2"
  ```

- **reset firefox:** to avoid interference between navigation of different websites, after each one, all the environment of the browser is cleaned. All the *tabs* are closed and private data including cache is eliminated. Everything is done simulating a series of touch events on proper coordinates on the display. Plus, the browser activity is closed from the multitasking menu every visit and once every 200 websites elaborated, Firefox is closed (and killed) through the proper option in the browser menu. The following code was developed for the OnePlus One. For the Nexus 5X, it remains the same but coordinates slightly change:

```
if model == 'opo':

   if it >= 200:
      it = 0
      sendTouchOPO(1000, 155) # open firefox menu
      sendTouchOPO(400,1740) # 'Quit Firefox'
      time.sleep(8)
   else:
      sendTouchOPO(900,163) # open tabs view
      sendTouchOPO(1024,152) # open tabs menu
      sendTouchOPO(634,288) # 'close all'
      sendTouchOPO(1000, 155) # open firefox menu
      sendTouchOPO(656,1437) # settings
      sendTouchOPO(240,1869) # clear private data
      sendTouchOPO(773,1760) # clean all

   subprocess.run(killfirefox.split())
   sendMenuOPO() # open multitasking view
   subprocess.run(("adb -s "+serial+" shell input swipe 600 200 600 1500
       50").split(" ")) # croll view of the multitasking menu
   sendTouchOPO(950,160) # kill all

   it = it + 1
```

- **log permission acceptance failures:** when `pos` or `vib` variables are set to `True`, it means that current websites exploit functions that are permission protected and that script has to accept. If it fails in granting them, there will be no trace of system calls in the logs. So the software calls the functions *checkLocationCall* and *checkVibrationCall* which read if the system calls are logged in the file and return the result. At this point, the script prints the name of the website in the *logfiles* described in previous section. This is the code:

```
if pos and not checkLocationCall():
   failedgrantingpos.write(str((i*4)+1)+","+row[1]+"\n")
if vib and not checkVibrationCall():
   failedgrantingvib.write(str((i*4)+1)+","+row[1]+"\n")
```

A.2.3 Functions

In this section are described all the functions exploited to do what was described in previous section.

cleanLogcat() The function in Fig. A.3 flushes all the *logs* on the device through `adb logcat -b all -c` command. Before executing that command, the `logcat` command is killed to avoid race conditions that would have prevented the flushing operation. After that, the logging process is restarted and the first two lines are extracted and compared to ones saved by the same function for the previous website. If they correspond it means that the cleaning command had failed and function returns `False`. On the other hand if it is successful the new first lines are saved in global variable `firstrowlog` and it returns `True`.

checkCurrentDomain() The function in Fig. A.4 is used to retrieve the domain currently visited by the website. It reads the logfile that is being written and through the *bash* text manipulation functions the last line containing *REDIRECTIONTO* tag is extracted. The split function to isolate the URL from the line is in a `try` block so that in the eventually no line is extracted, meaning the no website is loaded, we can be aware of the failure returning `None` string. If a URL is found but is the same from previous website, it means that website is not loaded as well and `None` is returned. On the other hand, if the extraction is successful, the current domain is returned.

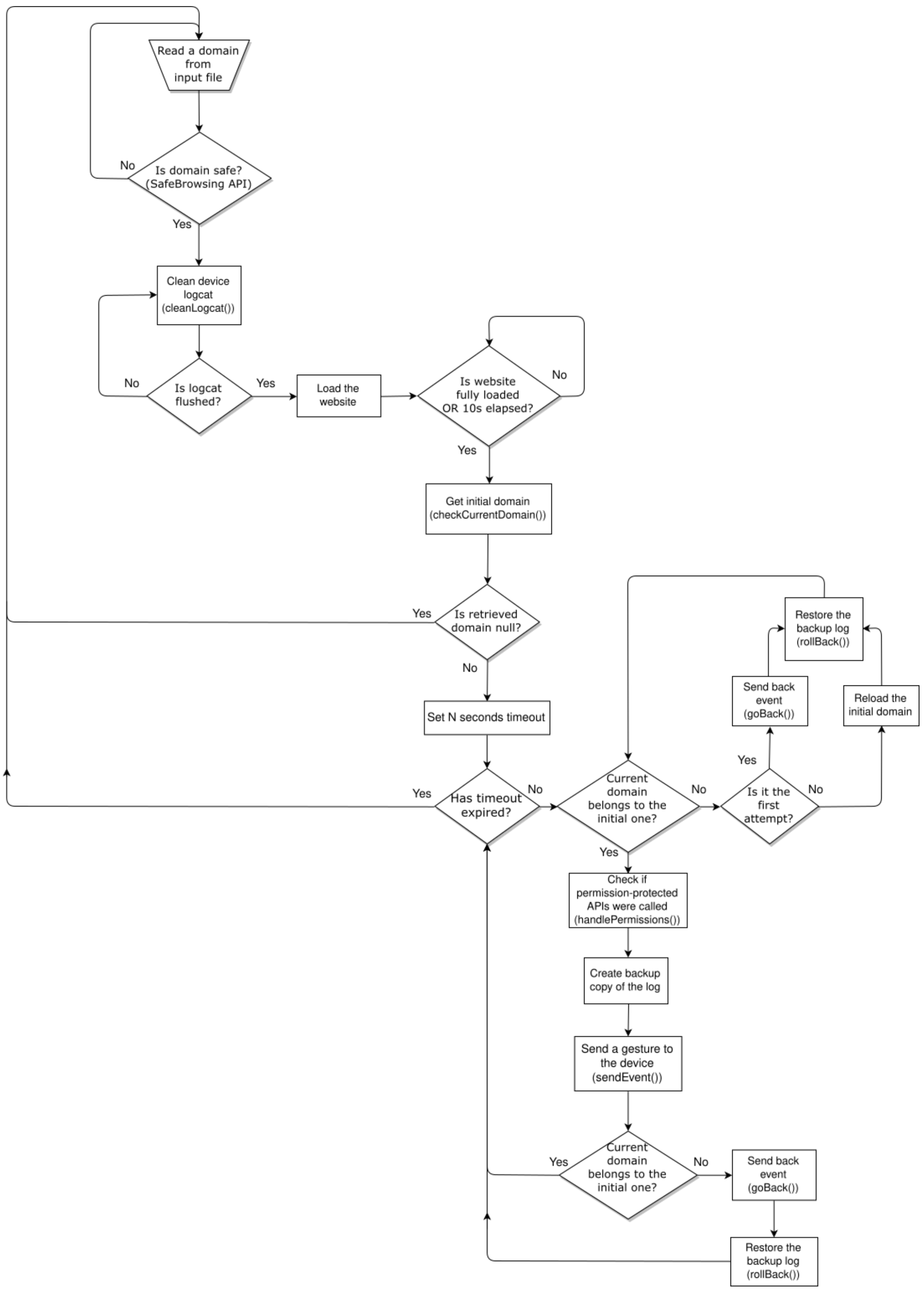

Read a domain from input file
Is domain safe? (SafeBrowsing API)
No
Yes
Clean device logcat (cleanLogcat())
Is logcat flushed?
No
Yes
Load the website
Is website fully loaded OR 10s elapsed?
No
Yes
Get initial domain (checkCurrentDomain())
Is retrieved domain null?
Yes
No
Set N seconds timeout
Has timeout expired?
Yes
No
Current domain belongs to the initial one?
No
Yes
Is it the first attempt?
Yes
No
Send back event (goBack())
Restore the backup log (rollBack())
Reload the initial domain
Check if permission-protected APIs were called (handlePermissions())
Create backup copy of the log
Send a gesture to the device (sendEvent())
Current domain belongs to the initial one?
Yes
No
Send back event (goBack())
Restore the backup log (rollBack())

Figure A.1. Flowchart of ffautomator script

checkPermissionRequests() The function in Fig. A.5 is used to check if the current visited website has requested permission to executed permission protected functions. They are the ones related to GPS position, vibration and media sources (cameras and microphone). Permissions popup is requested by the browser as soon as the function is called. The logs related to those particular calls have the special tag `MWEBAPILOGP`. Through the *bash* manipulation functions the last call with that tag is retrieved and based on the function contained in the log the returned string is different. Possible returned values are `position`, `vibration` and `media`. If an exception is raised during extraction, it means that no permission protected calls have been detected and `nothing` is returned. Plus, to avoid that the same call is detected more than one time making the script accept permission that are not requested, a dummy string is printed in the log with `DISCMWEBAPILOGP` tag. If the function extract this string, `nothing` will be returned as well.

handlePermissions() The function in Fig. A.6 is used to accept permission whether they are requested by the website. It calls the `checkPermissionRequests` function previously described to see if it is necessary to accept a permission request. If it is so, a sequence of *taps* is sent to the device at the coordinates of the popup button. A single tap is not enough because, while the width of the popup is always the same, the height changes depending on the length of the domain that is printed in the popup. Plus, the coordinates change depending on the model of the device and on which permission is requested. Once the permission is accepted, the related variable is set to `True`. This variable will be used in functions (`checkLocationCall` and `checkVibrationCall`) to check if the related system calls is executed at system level. Functions used to send gestures will be discussed later in this section.

checkLoading() The function in Fig. A.7 is used to check if website is fully loaded. The code injected in each website prints in the log a string with `FULLYLOADED` tag, when the events corresponding to the completed loading of the page is fired. Through the *bash* manipulation functions the we search a line containing that tag in the current *logfile*. If it is found the function returns `True`, otherwise an exception is raised and the returned value is `False`.

sendTouchOPO(x,y) and sendTouchNex(x,y) These functions are used to send a *tap* event to the device. It was necessary to create two of them because of the difference between the Android devices. Each model needs its own function. Functions receive as arguments the coordinates where the touch gesture has to happen and then sends a sequence of events to the special file at system level that controls the touchscreen. In Fig. A.8 it is shown the code related to the OnePlus One model, for others the function would be the same, except for the arguments in `sendevent` functions. A guide on how to realize the chain of events to create a gesture can be found at `https://qatesttech.wordpress.com/2012/06/20/adb-shell-sendevent-sending-touch-like-events/`.

sendSwipeOPO(x,y) and sendSwipeNex(x,y) These functions are the analogous of `sendTouchOPO(x,y)` and `sendTouchNex(x,y)` but to send *swipe* gestures. As for the other functions, in Fig. A.9 we report only the code for the OnePlus One. Differently for the functions to send *taps*, now the coordinates are not passed as arguments but are randomly generated by the function itself. And we need two couples of coordinates, the first representing the *starting point* and the second the *ending point*. Sending swipes requires also the length of the gestures, that is the number of steps necessary to reach the end point from the starting one. to make all swipes last the same, the number of steps must be proportioned to the distance between the two endpoints of the gestures. At the beginning of the function we calculate that distance and based on the result we divide it in a different number of steps, retrieving `dx` and `dy` that are the lengths of each step on the two axis.

sendEvent() The function in Fig. A.10 is used to randomly send a touch gesture to the device. It is composed by two identical parts, one for each device used, because based on the model the functions to send gestures change. In the script we need to simulate navigation in websites through random events, so the type of the gesture that is sent (*tap* or *swipe*) is randomly extracted. On the other hand we wanted to have control on the percentage of the event sent because for navigation we thought that *taps* were more effective than *swipes*. For this reason we extract a number out of 4, and we send a swipe only if it is 0, otherwise we sent a *tap*. With this strategy, the 75%

of events sent are *taps*. This percentage can be easily modified, changing the parameters of the random extraction.

checkLocationCall() and checkVibrationCall() These functions are used to check whether permissions are accepted for permissions protected calls. We know that if permissions are accepted, the related system-calls are generated by the OS. We identified them and so we look for them in the generated *logfile* using *bash* text manipulation commands. If in the extraction of them an exception is raised, it means that system calls have not been executed and permissions was not correctly accepted. In this case function return `False`, otherwise `True`. In Fig. A.11 it is shown the code of `checkLocationCall`, it differs from `checkVibratioCall` for the name of the system call that is searched in the file and for controlling if the website provides a secure connection (*HTTPS*). This check is necessary because if the website relies on a insecure connection, the position is not retrieved by the phone and, even if the permission is granted, the data is not gathered.

rollBack() The function in Fig. A.12 is used to restore the correct version of the *log* that was saved as a backup, in case the navigation is redirected to an external domain. The function exploits *bash* commands to copy back the backup copy overwriting the current faulty one and then it logs on the file the action indicating the two involved domains.

```
if currentdomain in initialdomain or initialdomain in currentdomain:

    time.sleep(0.2)

    createcopy = ("cp logs"+serial+"/"+row[1]+".txt
        logs"+serial+"/"+row[1]+"2.txt")
    subprocess.run(createcopy.split(" "))

    handlePermissions() #function handling permissions acceptance
    sendEvent() #function sending gesture

    time.sleep(0.2)
    currentdomain = checkCurrentDomain()
else:

    goBack()
    print("ROLLBACK1")
    rollBack()

    time.sleep(0.7)

    currentdomain = checkCurrentDomain()

    if currentdomain in initialdomain or initialdomain in currentdomain:

       createcopy = ("cp logs"+serial+"/"+row[1]+".txt
           logs"+serial+"/"+row[1]+"2.txt")
       subprocess.run(createcopy.split(" "))
       time.sleep(1)
       handlePermissions() #function handling permissions acceptance
       sendEvent() #function sending gesture
       time.sleep(0.3)
       currentdomain = checkCurrentDomain()

    else: # if after going back the domain is still different reload the
        initial website

       subprocess.run(command.split(" "))
       time.sleep(2)

       rollBack()

       dummyred = "adb -s " + serial + " shell log -p v -t Dummylog \"DUMMY
           REDIRECTIONTO: " + str(initialdomain) + "\"" #dummy redirection
           log to avoid that a slowcompromise the program the addess bar is
           not written immediatly)
       subprocess.run(dummyred.split(" "))
       time.sleep(0.7)
       handlePermissions()
       sendEvent()
       time.sleep(0.3)
       currentdomain = checkCurrentDomain()
```

Figure A.2. This is the code responsible to handle the rollback to the correct domain

```
def cleanLogcat():
   global firstrowlog
   subprocess.run(("adb -s "+serial+" shell su -c \'pkill logcat\'").split("
       ")) # kills all processes using logcat to avoid cleaning race
       conditions
   subprocess.run(("adb -s "+serial+" logcat -b all -c").split(" "))
   p1 = subprocess.Popen(["adb", "-s", serial, "logcat", "-v", "time"],
       stdout=subprocess.PIPE)
   first = subprocess.check_output(["head", "-n", "2"], stdin=p1.stdout)
   p1.stdout.close()
   p1.terminate()
   if firstrowlog in str(first):
      return False
   else:
      firstrowlog = str(first)
      return True
```

Figure A.3. cleanLogcat function

```
def checkCurrentDomain():
   p1 = subprocess.Popen(["cat", "logs"+serial+"/" + row[1] + ".txt"],
       stdout=subprocess.PIPE)
   p2 = subprocess.Popen(["grep", "-a","REDIRECTIONTO"], stdin=p1.stdout,
       stdout=subprocess.PIPE)
   url = subprocess.check_output(["tail", "-n", "1"], stdin=p2.stdout)
   p1.stdout.close()
   p2.stdout.close()
   p1.terminate()
   p2.terminate()
   try:
      domain = str(url).split(": ")[2][:-3]
      if domain == domainold:
         print("----------> DOMAIN is the last one, not saved :
             ["+domainold+"] <------------\n")
         return "None"
      else:
         print("CURRENT DOMAIN --------------> " + str(url).split(":
             ")[2][:-3] + "<--------------------")
         return domain
   except:
      print("ERROR in REDIRECTION DETECTION (NOT FOUND)")
      return "None"
```

Figure A.4. checkCurrentDomain function

```
def checkPermissionRequests():
   p1 = subprocess.Popen(["cat", "logs"+serial+"/" + row[1] + ".txt"],
       stdout=subprocess.PIPE)
   p2 = subprocess.Popen(["grep", "-a","MWEBAPILOGP;"], stdin=p1.stdout,
       stdout=subprocess.PIPE)
   permission = subprocess.check_output(["tail", "-n", "1"], stdin=p2.stdout)
   p1.stdout.close()
   p2.stdout.close()
   p1.terminate()
   p2.terminate()
   try:
      domain = str(permission).split(";")[1]
      # options are: vibrate - getCurrentPosition - watchPosition -
          getUserMedia
      # a dummy log 'DISCMWEBAPILOGP' is written to avoid next accpeting
          conditions on calls already processed
      # if the dummy log is detected nothing happens
      if domain == "vibrate":
         logprint = "adb -s " + serial + " shell log -p v -t Dummylog
             \"DISCMWEBAPILOGP;nothing\""
         subprocess.run(logprint.split(" "))
         return "vibrate"
      elif domain == "getCurrentPosition" or domain == "watchPosition":
         logprint = "adb -s "+ serial +" shell log -p v -t Dummylog
             \"DISCMWEBAPILOGP;nothing\""
         subprocess.run(logprint.split(" "))
         return "position"
      elif domain == "getUserMedia":
         logprint = "adb -s " + serial +" shell log -p v -t Dummylog
             \"DISCMWEBAPILOGP;nothing\""
         subprocess.run(logprint.split(" "))
         return "media"
      else:
         return "nothing"
   except Exception as e:
      print("Can't retrieve any permission protected WEBAPI call, "+str(e))
      return "nothing"
```

Figure A.5. checkPermissionRequests function

```
def handlePermissions():
    global pos
    global vib

    perm = checkPermissionRequests()
    print("--------> " + perm)
    if perm == "vibrate":
        time.sleep(0.5)
        if model == 'opo':
            sendTouchOPO(970, 400)
            sendTouchOPO(970, 460)
            sendTouchOPO(970, 520)
            sendTouchOPO(970, 600)
            sendTouchOPO(970, 700)
            sendTouchOPO(970, 800)
            sendTouchOPO(970, 900)
        else:
            sendTouchNex(920, 400)
            sendTouchNex(920, 460)
            sendTouchNex(920, 520)
            sendTouchNex(920, 550)
            sendTouchNex(920, 600)
        vib=True
        print("PERMISSION vibrate accepted")
    elif perm == "position":
        time.sleep(0.5)
        if model == 'opo':
            sendTouchOPO(980, 330)
            ...
            sendTouchOPO(970, 900)
        else:
            sendTouchNex(920, 330)
            ...
            sendTouchNex(920, 730)
        pos=True
        print("PERMISSION position accepted")
    elif perm == "media":
        time.sleep(0.5)
        if model == 'opo':
            sendTouchOPO(980, 440)
            ...
            sendTouchOPO(980, 1340)
        else:
            sendTouchNex(920, 430)
            ...
            sendTouchNex(920, 1330)

        print("PERMISSION media accepted")
    else:
        print("NO PERMISSION REQUESTED")
```

Figure A.6. handlePermissions function (parts of code that are analogous to the previous ones have been hidden)

```
def checkLoading():
    p1 = subprocess.Popen(["cat", "logs"+serial+"/" + row[1] + ".txt"],
        stdout=subprocess.PIPE)
    p2 = subprocess.Popen(["grep", "-a","FULLYLOADED"], stdin=p1.stdout,
        stdout=subprocess.PIPE)
    loaded = subprocess.check_output(["tail", "-n", "1"], stdin=p2.stdout)
    p1.stdout.close()
    p2.stdout.close()
    p1.terminate()
    p2.terminate()
    try:
        l = str(loaded)
        if "FULLYLOADED" in l:
            return True
        else:
            return False

    except Exception as e:
        return False
```

Figure A.7. checkLoading function

```
def sendTouchOPO(x,y):
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 3 57
        153").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        330 1").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        325 1").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 3 53
        "+str(x)).split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 3 54
        "+str(y)).split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 3 58
        168").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 0 0
        0").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 3 57
        -1").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        330 0").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        325 0").split(" "))
    subprocess.run(("adb -s "+serial+" shell sendevent /dev/input/event0 0 0
        0").split(" "))
```

Figure A.8. sendTouchOPO function

```
def sendSwipeOPO():

    x1 = randomx()
    x2 = randomx()
    y1 = randomy()
    y2 = randomy()

    d=math.sqrt((x2-x1)**2+(y2-y1)**2)
    if d < 150:
        steps = 2
    elif d < 280:
        steps = 2
    elif d < 550:
        steps = 2
    elif d < 1100:
        steps = 3
    else:
        steps = 4
    xt = x1
    yt = y1
    dx = int((x2-x1)/steps)
    dy = int((y2-y1)/steps)

    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 3
        57 9").split(" "))
    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        330 1").split(" "))
    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        325 1").split(" "))

    for _ in range(0,steps):
        subprocess.Popen(("adb -s "+serial+" shell sendevent
            /dev/input/event0 3 53 "+str(xt)).split(" "))
        subprocess.Popen(("adb -s "+serial+" shell sendevent
            /dev/input/event0 3 54 "+str(yt)).split(" "))
        subprocess.Popen(("adb -s "+serial+" shell sendevent
            /dev/input/event0 3 58 188").split(" "))
        subprocess.Popen(("adb -s "+serial+" shell sendevent
            /dev/input/event0 0 0 0").split(" "))
        xt = xt + dx
        yt = yt + dy
    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 3
        57 -1").split(" "))
    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        330 0").split(" "))
    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 1
        325 0").split(" "))
    subprocess.Popen(("adb -s "+serial+" shell sendevent /dev/input/event0 0
        0 0").split(" "))
```

Figure A.9. sendSwipeOPO function

```
def sendEvent():
    if model == 'opo':
        if random.randint(0,4) > 0:
            x1 = randomx()
            y1 = randomy()
            sendTouchOPO(x1, y1)
        else:
            sendSwipeOPO()
    else:
        if random.randint(0,4) > 0:
            x1 = randomx()
            y1 = randomy()
            sendTouchNex(x1, y1)
        else:
            sendSwipeNex()
```

Figure A.10. sendEvent function

```
def checkLocationCall():
    p1 = subprocess.Popen(["cat", "logs"+serial+"/" + row[1] + ".txt"],
        stdout=subprocess.PIPE)
    p2 = subprocess.Popen(["grep", "-a","getLastKnownLocation"],
        stdin=p1.stdout, stdout=subprocess.PIPE)
    calll = subprocess.check_output(["tail", "-n", "1"], stdin=p2.stdout)
    p1.stdout.close()
    p2.stdout.close()
    p1.terminate()
    p2.terminate()
    try:
        call = str(calll)
        print(call)
        if 'getLastKnownLocation' in call:
            return True
        else:
            p1 = subprocess.Popen(["cat", "logs"+serial+"/" + row[1] +
                ".txt"], stdout=subprocess.PIPE)
            p2 = subprocess.Popen(["grep", "-a", "\"A Geolocation request can
                only be fulfilled in a secure context\""], stdin=p1.stdout,
                stdout=subprocess.PIPE)
            errr = subprocess.check_output(["tail", "-n", "1"],
                stdin=p2.stdout)
            try:
                err = str(errr)
                print("EXC2: "+err)
                if 'fulfilled' in err:
                    return True
            except Exception as e:
                return False
            return False
    except Exception as e:
        print("EXC: "+str(e))
        return False
```

Figure A.11. checkLocationCall function

```
def rollBack():
    global initialdomain
    global currentdomain
    print("Rollback backup LOG")
    rollbacklog.write(str(datetime.datetime.now()) +" Rollback needed ---"
        +str(initialdomain) +" ---> " +str(currentdomain)+" ["+row[1]+"]\n")
    rollback = "cp logs"+serial+"/"+row[1]+"2.txt
        logs"+serial+"/"+row[1]+".txt"
    #print(str(datetime.datetime.now()) + "rollback logs")
    logrollback = "adb -s "+serial+" shell log -p v -t ROLLBACK
        \"init:"+str(initialdomain)+ " curr: "+str(currentdomain)+"\""
    subprocess.run(rollback.split(" "))
    subprocess.run(logrollback.split(" "))
    currentdomain = initialdomain
```

Figure A.12. rollBack function

Bibliography

[1] A. Lella, "U.S. Smartphone Penetration Surpassed 80 Percent in 2016." https://www.comscore.com/Insights/Blog/US-Smartphone-Penetration-Surpassed-80-Percent-in-2016, Accessed: 2018-04-18

[2] Global Stats, "Mobile and tablet internet usage exceeds desktop for first time worldwide." http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide, Accessed: 2018-04-19

[3] G. Sterling, "Mobile Devices Now Driving 56 Percent Of Traffic To Top Sites." https://marketingland.com/mobile-top-sites-165725, Accessed: 2018-10-13

[4] Fazal-e-Amin, "Characterization of web browser usage on smartphones", Computers in human behavior, vol. 51, October 2015, pp. 896–902, DOI 10.1016/j.chb.2014.10.054

[5] Google Developers, "Request App Permissions." https://developer.android.com/training/permissions/requesting, Accessed: 2018-09-15

[6] M. Firtman, "Mobile HTML5 Compatibility on Mobile Devices." http://mobilehtml5.org/, Accessed: 2018-04-22

[7] A. Zimba, Z. Wang, M. Mulenga, and N. H. Odongo, "Crypto mining attacks in information systems: An emerging threat to cyber security", Journal of Computer Information Systems, May 2018, pp. 1–12, DOI 10.1080/08874417.2018.1477076

[8] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser analysis of web-based malware", First Conference on First Workshop on Hot Topics in Understanding Botnets, Berkeley, CA, USA, 2007, pp. 4–4

[9] G. Greenwald, "No place to hide: Edward snowden, the NSA, and the U.S. surveillance state", Macmillan, May 2014, ISBN: 9781627790734

[10] Y. Wu, D. Meng, and H. Chen, "Evaluating private modes in desktop and mobile browsers and their resistance to fingerprinting", 2017 IEEE Conference on Communications and Network Security (CNS), October 2017, pp. 1–9, DOI 10.1109/CNS.2017.8228636

[11] C. Warren, E. El-Sheikh, and N.-A. Le-Khac, "Privacy preserving internet browsers: Forensic analysis of browzar", Computer and Network Security Essentials (K. Daimi, ed.), pp. 369–388, Cham: Springer International Publishing, 2018, DOI 10.1007/978-3-319-58424-9_21

[12] P. Laperdrix, "Browser fingerprinting: Exploring device diversity to augment authentification and build client-side countermeasures". PhD thesis, Rennes, INSA, 2017

[13] L. Olejnik, C. Castelluccia, and A. Janc, "On the uniqueness of web browsing history patterns", annals of telecommunications - annales des télécommunications, February 2014, pp. 63–74, DOI 10.1007/s12243-013-0392-5

[14] D. Cameron, "Apple Declares War on 'Browser Fingerprinting,' the Sneaky Tactic That Tracks You in Incognito Mode." https://gizmodo.com/apple-declares-war-on-browser-fingerprinting-the-sneak-1826549108, Accessed: 2018-09-18

[15] T. Hupperich, D. Maiorca, M. Kührer, T. Holz, and G. Giacinto, "On the robustness of mobile device fingerprinting: Can mobile users escape modern Web-Tracking mechanisms?", 31st Annual Computer Security Applications Conference, New York, NY, USA, 2015, pp. 191–200, DOI 10.1145/2818000.2818032

[16] Ł. Olejnik, G. Acar, C. Castelluccia, and C. Diaz, "The leaking battery", Data Privacy Management, and Security Assurance (J. Garcia-Alfaro, G. Navarro-Arribas, A. Aldini, F. Martinelli, and N. Suri, eds.), vol. 9481 of *Lecture Notes in Computer Science*, pp. 254–263,

Cham: Springer International Publishing, 2016, DOI 10.1007/978-3-319-29883-2_18

[17] A. Das, N. Borisov, and E. Chou, “Every move you make: Exploring practical issues in smartphone motion sensor fingerprinting and countermeasures”, Proceedings on Privacy Enhancing, 2018, pp. 88–108, DOI 10.1515/popets-2018-0005

[18] P. Snyder, C. Taylor, and C. Kanich, “Most websites Don’T need to vibrate: A Cost-Benefit approach to improving browser security”, 2017 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 2017, pp. 179–194, DOI 10.1145/3133956.3133966

[19] C. C. Tossell, “An empirical analysis of internet use on smartphones: characterizing visit patterns and user differences”. PhD thesis, Rice University, 2012

[20] L. Olejnik, “Report on sensors APIs: privacy and transparency perspective.” `https://lukaszolejnik.com/SensorsPrivacyReport.pdf`, Accessed: 2018-09-30

[21] rovo89, “Xposed framework.” `https://repo.xposed.info`, Accessed: 2018-06-14

[22] A. Cortesi, M. Hils, and T. Kriechbaumer, “mitmproxy.” `https://mitmproxy.org`, v. 3.0.3

[23] H. N. Anh, “Smartphone industry: The new era of competition and strategy”, bachelor’s thesis, Centria University of Applied Sciences, 2016

[24] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones”, NDSS Symposium 2012, February 2012, p. 19

[25] W. Enck, D. Octeau, P. Mcdaniel, and S. Chaudhuri, “A study of android application security”, In Proc. USENIX Security Symposium, 2011

[26] P. Laperdrix, W. Rudametkin, and B. Baudry, “Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints”, 2016 IEEE Symposium on Security and Privacy (SP), May 2016, pp. 878–894, DOI 10.1109/SP.2016.57

[27] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, “I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis”, CoRR, vol. abs/1404.7431, 2014

[28] C. Gibler, J. Crussell, J. Erickson, and H. Chen, “AndroidLeaks: Automatically detecting potential privacy leaks in android applications on a large scale”, Trust and Trustworthy Computing (S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, eds.), vol. 7344 of *Lecture Notes in Computer Science*, pp. 291–307, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, DOI 10.1007/978-3-642-30921-2_17

[29] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An Information-Flow tracking system for realtime privacy monitoring on smartphones”, ACM Transactions on Computer Systems, vol. 32, June 2014, pp. 5:1–5:29, DOI 10.1145/2619091

[30] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, “AppIntent: analyzing sensitive data transmission in android for privacy leakage detection”, 2013 ACM SIGSAC conference on Computer & communications security, New York, NY, USA, 2013, pp. 1043–1054, DOI 10.1145/2508859.2516676

[31] J. Seo, D. Kim, D. Cho, T. Kim, and I. Shin, “FLEXDROID: Enforcing In-App privilege separation in android”, Network and Distributed System Security Symposium, Reston, VA, 2016, DOI 10.14722/ndss.2016.23485

[32] Y. Zhauniarovich and O. Gadyatskaya, “Small changes, big changes: An updated view on the android permission system”, Research in Attacks, Intrusions, and Defenses, 2016, pp. 346–367, DOI 10.1007/978-3-319-45719-2_16

[33] M. Diamantaris, E. Papadopoulos, E. Markatos, S. Ioannidis, and J. Polakis, “Real-time app analysis for augmenting the android permission system”, (under submission)

[34] H. Shahriar, T. Klintic, and V. Clincy, “Mobile phishing attacks and mitigation techniques”, Journal of Information Security, vol. 6, no. 3, 2015, pp. 206–212, DOI 10.4236/jis.2015.63021

[35] L. Wu, X. Du, and J. Wu, “MobiFish: A lightweight anti-phishing scheme for mobile phones”, 2014 23rd International Conference on Computer Communication and Networks (ICCCN), August 2014, pp. 1–8, DOI 10.1109/ICCCN.2014.6911743

[36] S. J. Tripathi, V. S. Gangwani, and M. E. Student, “Design the framework for detecting malicious mobile webpages in real time”, International Research Journal of Engineering and Technology, vol. 4, February 2017, pp. 441–444

[37] N. M. Al-Fannah, “One leak will sink a ship: WebRTC IP address leaks”, 2017 International Carnahan Conference on Security Technology (ICCST), October 2017, pp. 1–5, DOI

10.1109/CCST.2017.8167801

[38] A. Kostiainen and M. Lamouri, “Battery status API.” https://www.w3.org/TR/battery-status/, Accessed: 2018-07-20

[39] L. Olejnik, “Battery Status readout as a privacy risk.” https://blog.lukaszolejnik.com/battery-status-readout-as-a-privacy-risk/, Accessed: 2018-07-20

[40] C. Cimpanu, “Battery Status API being Removed from Firefox due to Privacy Concerns.” https://www.w3.org/TR/battery-status/, Accessed: 2018-07-20

[41] W. Jobe, “Native apps vs. mobile web apps”, International Journal of Interactive Mobile, vol. 7, October 2013, pp. 27–32, DOI 10.3991/ijim.v7i4.3226

[42] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos, “The Long-Standing privacy debate: Mobile websites vs mobile apps”, 26th International Conference on World Wide Web, April 2017, pp. 153–162, DOI 10.1145/3038912.3052691

[43] A. K. Ratha, S. Sahu, and P. Meher, “HTML5 in web development: A new approach”, International Research Journal of Engineering and Technology (IRJET), vol. 5, March 2018, pp. 551–554

[44] Y. Maheshwari and Y. R. Reddy, “A study on migrating flash files to HTML5/JavaScript”, 10th Innovations in Software Engineering Conference, February 2017, pp. 112–116, DOI 10.1145/3021460.3021472

[45] A. Kostiainen, “Ambient light sensor API.” https://www.w3.org/TR/ambient-light/, Accessed: 2018-07-13

[46] Anssi Kostiainen, “Vibration API.” https://www.w3.org/TR/vibration/, Accessed: 2018-07-13

[47] R. Tibbett, T. Volodine, S. Block, and A. Popescu, “Device orientation event.” https://www.w3.org/TR/orientation-event/, Accessed: 2018-07-13

[48] M. Lamouri and M. Cáceres, “Screen orientation API.” https://www.w3.org/TR/screen-orientation/, Accessed: 2018-07-13

[49] A. Kostiainen and R. Bhaumik, “Proximity sensor API.” https://www.w3.org/TR/proximity/, Accessed: 2018-07-13

[50] D. C. Burnett, A. Bergkvist, C. Jennings, A. Narayanan, and B. Aboba, “Media capture API.” https://www.w3.org/TR/mediacapture-streams/, Accessed: 2018-07-13

[51] A. Abdulmunim, “Mobile web browsers in android deriving reference architecture”, IJCAI: proceedings of the conference / sponsored by the International Joint Conferences on Artificial Intelligence, vol. 180, January 2018, pp. 17–22, DOI 10.5120/ijca2018916284

[52] E. Janczukowicz, “Firefox os overview”. PhD thesis, Télécom Bretagne, 2013

[53] G. Dong, Y. Zhang, X. Wang, P. Wang, and L. Liu, “Detecting cross site scripting vulnerabilities introduced by HTML5”, 2014 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), May 2014, pp. 319–323, DOI 10.1109/JCSSE.2014.6841888

[54] P. Eckersley, “How unique is your web browser?”, Privacy Enhancing Technologies (M. J. Atallah and N. J. Hopper, eds.), vol. 6205 of *Lecture Notes in Computer Science*, pp. 1–18, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, DOI 10.1007/978-3-642-14527-8_1

[55] Valentin Vasilyev, “Modern & flexible browser fingerprinting library.” https://github.com/Valve/fingerprintjs2, Accessed: 2018-09-22

[56] G. Nakibly, G. Shelef, and S. Yudilevich, “Hardware fingerprinting using HTML5”, arXiv preprint arXiv:1503.01408, March 2015

[57] R. Upathilake, Y. Li, and A. Matrawy, “A classification of web browser fingerprinting techniques”, 2015 7th International Conference on New Technologies, Mobility and Security (NTMS), July 2015, pp. 1–5, DOI 10.1109/NTMS.2015.7266460

[58] P. Snyder, L. Ansari, C. Taylor, and C. Kanich, “Browser feature usage on the modern web”, 2016 Internet Measurement Conference, November 2016, pp. 97–110, DOI 10.1145/2987443.2987466

[59] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, “Using mobile phones to determine transportation modes”, ACM Trans. Sen. Netw., vol. 6, March 2010, pp. 13:1–13:27, DOI 10.1145/1689239.1689243

[60] Z. Xu, K. Bai, and S. Zhu, “TapLogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors”, Fifth ACM Conference on Security and Privacy in Wireless and

Mobile Networks, New York, NY, USA, 2012, pp. 113–124, DOI 10.1145/2185448.2185465

[61] J. Han, E. Owusu, L. T. Nguyen, A. Perrig, and J. Zhang, "ACComplice: Location inference using accelerometers on smartphones", 2012 Fourth International Conference on Communication Systems and Networks (COMSNETS 2012), January 2012, pp. 1–9, DOI 10.1109/COMSNETS.2012.6151305

[62] X. Bai, J. Yin, and Y.-P. Wang, "Sensor guardian: prevent privacy inference on android sensors", EURASIP Journal on Information Security, vol. 2017, June 2017, p. 10, DOI 10.1186/s13635-017-0061-8

[63] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block me if you can: A Large-Scale study of Tracker-Blocking tools", 2017 IEEE European Symposium on Security and Privacy (EuroSP), April 2017, pp. 319–333, DOI 10.1109/EuroSP.2017.26

[64] O. Starov and N. Nikiforakis, "PrivacyMeter: Designing and developing a Privacy-Preserving browser extension", Engineering Secure Software and Systems (M. Payer, A. Rashid, and J. M. Such, eds.), Cham, 2018, pp. 77–95, DOI 10.1007/978-3-319-94496-8_6

[65] O. A. V. Ravnås, "Frida." `https://frida.re`, Accessed: 2018-06-14

[66] B. Alman (cowboy), "Monkey-patch (hook) functions for debugging and stuff." `https://github.com/cowboy/javascript-hooker`, Accessed: 2018-04-23

[67] ghostwords, "Browser fingerprinting protection for everybody." `https://github.com/ghostwords/chameleon`, Accessed: 2018-06-09

[68] L.-M. Keinånen, "Touch screen mobile devices invading the internet: Ux guidelines towards one web", Master's thesis, Aalto University, 2011

[69] S. S Machiraju, A. K Athukuri, S. Gampa, N. B Makela, and V. N Inukollu, "Application based smart optimized keyboard for mobile apps", Computer Science & Information Technology (CS & IT), January 2017, pp. 175–186, DOI 10.5121/csit.2017.70117

[70] F. Khomh, H. Yuan, and Y. Zou, "Adapting linux for mobile platforms: An empirical study of android", 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 629–632, DOI 10.1109/ICSM.2012.6405339

[71] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis", 2016 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 2016, pp. 1388–1401, DOI 10.1145/2976749.2978313