

Research Article

Session Fingerprinting in Android via Web-to-App Intercommunication

Efthimios Alepis and Constantinos Patsakis 

Department of Informatics, University of Piraeus, 80 Karaoli & Dimitriou, 18534 Piraeus, Greece

Correspondence should be addressed to Constantinos Patsakis; kpatsak@gmail.com

Received 29 December 2017; Accepted 3 June 2018; Published 28 June 2018

Academic Editor: Guojun Wang

Copyright © 2018 Efthimios Alepis and Constantinos Patsakis. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The extensive adoption of mobile devices in our everyday lives, apart from facilitating us through their various enhanced capabilities, has also raised serious privacy concerns. While mobile devices are equipped with numerous sensors which offer context-awareness to their installed apps, they can also be exploited to reveal sensitive information when correlated with other data or sources. Companies have introduced a plethora of privacy invasive methods to harvest users' personal data for profiling and monetizing purposes. Nonetheless, up till now, these methods were constrained by the environment they operate, e.g., browser versus mobile app, and since only a handful of businesses have actual access to both of these environments, the conceivable risks could be calculated and the involved enterprises could be somehow monitored and regulated. This work introduces some novel user deanonymization approaches for device and user fingerprinting in Android. Having Android AOSP as our baseline, we prove that web pages, by using several inherent mechanisms, can cooperate with installed mobile apps to identify which sessions operate in specific devices and consequently further expose users' privacy.

1. Introduction

The unprecedented growth of mobile usage has radically transformed our daily lives. In addition to the great advances in our communications, mobile devices have changed the way we create, process, and consume information, as they realize pervasive and ubiquitous computing. Among others, one of the most significant emerged changes is how we value information. The fact that people are constantly and effortlessly connected to the Internet via smart devices which empower people's unobstructed communication, information flow, and entertainment in many occasions results in disregarding or underestimating the value of the information they consume and offer to third parties. This kind of collected data is considered as the world's new oil [1] but is also accompanied by an increased risk regarding users' privacy.

Subsequently, as far as information offering is concerned, the value of the provided information to third parties is in most of the cases considerably high, something that is not always understood by the users. For instance, one might share his location with an app or a web page neglecting the fact that

this single piece of information also encloses a very sensitive piece of data which can be exploited for various purposes. Indicative uses for such location sharing could be the recommendation of other users in proximity for communication purposes or even for sharing a ride. Aggregating location data from numerous users can provide real-time traffic analytics or insight into resource requirements in a smart city. Apparently, this information can stimulate businesses' prosperity by enabling the implementation of further customer-centered services. Therefore most companies are striving to extract as much information as possible from users.

While data mining offers undeniable advantages to users, e.g., service personalization can be considered as a noble cause, companies tend to exploit data even further for profiling and targeted advertising. Such tactics can expose users to many privacy hazards. This trend is highlighted by the fact that many companies are providing APIs which harvest user data to create fine-grained user profiles, containing a lot of sensitive user information. Such practices have also led to the introduction of methods such as browser and device fingerprinting. Nonetheless, mobile apps and web pages are

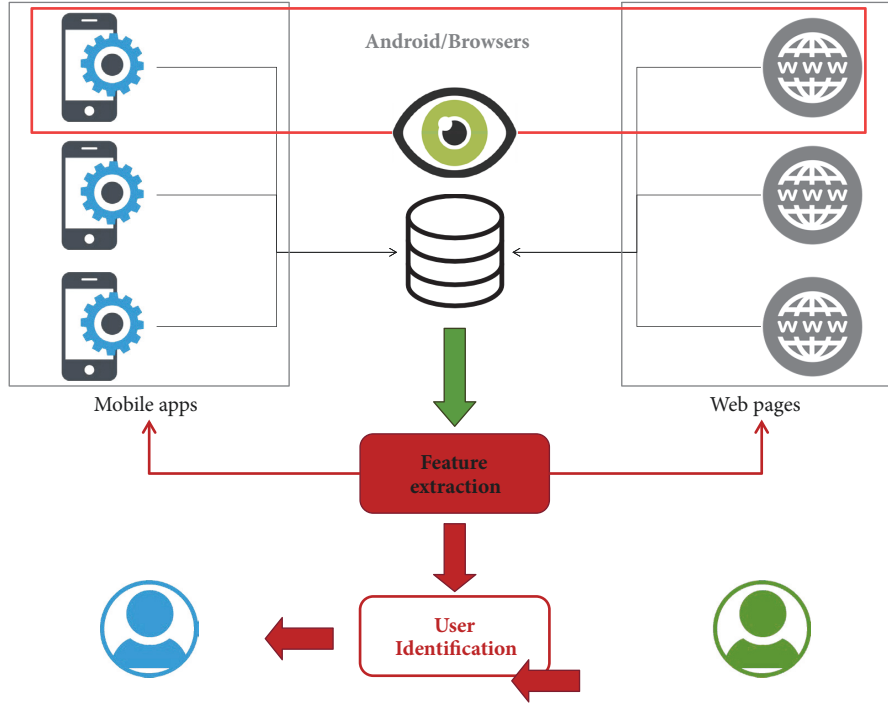


FIGURE 1: Basic concept.

thus far considered as two diverse ecosystems, as they refer to two discrete software environments with radical differences in their information flow and data usage. This distinction works in favor of users' privacy, since it allows some parts of their activities to remain isolated and hence private. For instance, it prevents an app from knowing which web pages a user visits or a web page from knowing which apps a user is using and when. On the contrary, enabling access between these two environments could allow for a web page to communicate with an installed app to recover further sensitive personal information from local files or sensor measurements and hence further reveal one's interests.

The goal of this work is to illustrate that there are currently several means to realize user identification in Android, regardless of the environment a software module is operating on. Despite the privacy hesitations that people might have towards the well-known tech giants or independent browsers, we provide some concrete examples proving that an "All Seeing Eye", a software entity able to monitor users' actions across both the web and the application ecosystems, can be easily created. Such an entity, in the form of an cloud-based database equipped with some additional services can correlate information from web pages and mobile apps in order to identify individuals. After a thorough investigation in the related scientific literature and to the best of our knowledge, the authors of this paper have concluded that this problem has been so far only partially studied, as current literature is focused on methods which examine each software ecosystem independently and not both of them as a whole. In fact, the proposed methods in this work can be considered as an extension of device fingerprinting as they

do not solely depend upon unique characteristics of device components or hardware identifiers. We label these methods as "session fingerprinting" since their goal is to reveal whether web-browsing and software sessions operate simultaneously in a device and identify the user.

The generic concept of this work, in a simplified form, is illustrated in Figure 1. Each side of this figure is dedicated to the two software "ecosystems", namely web pages and mobile apps. Obviously, there is a crosscut from the OS, namely, Android, since it manages calls from both ecosystems in a mobile device, as well as from the browsers which by definition belong as applications to both ecosystems. The "All Seeing Eye" acts as a Command and Control, C&C, server which collects information from web pages and apps, correlates it, and transmits "commands" and the corresponding information to both sides. The commands may range from "retrieve a list of installed apps" and "scan local storage for files containing X", to "display ad Y" or "application Z send webpage data W". Therefore, the "All Seeing Eye", as the orchestrator of all performed actions by apps and web pages can ultimately reveal user identities.

While similar attempts have been made in the past, it is rather important to note that methods trying to escape the browser's environment without users' consent are considered to be malware and usually exploit browsers' vulnerabilities. Especially in the case of Android, passing a single bit of information from a benign browser to an app is rather difficult, given the fact that it has to bypass not only the browser's sandbox but also additional obstacles due to Android's inherent security model which will be discussed later on.

This paper extends previous work of the authors [2] by providing more details for the underlying methods, the related literature, and also experiments regarding session fingerprinting. The rest of this work is organized as follows. In the next section we present the related work, discussing methods for user profiling in mobile devices and browsers and some Android specific details regarding permissions of apps. In Section 3, our newly introduced concept of “session fingerprinting” is analyzed and in Section 4 we state the problem we address and discuss how both apps and web pages are expected to behave in this context. In Section 5 we present four concrete examples which prove the efficacy of our approach and detail how they can be realized. Section 6 illustrates the extension of the threat by providing experimental result and statistics. Finally, we conclude discussing some of our findings and ideas for future research.

2. Related Work

2.1. Isolation of Apps in Android. Android started as a heavily modified Linux distribution to meet the needs of mobile devices which had significantly fewer resources than desktop computers. However, the introduced changes made it quite unique, leading many people to consider it something beyond a different Linux distribution.

Contrary to most operating systems, the actual user of the device does not have administrative privileges by default. While this choice is actually preventing the user to have complete control of the device he owns, it also prevents adversaries to gain more privileges than they should. Certainly, there are several attacks presented in the literature [3, 4]; however, they can be considered as few and quickly patched by Google. Since users tend to install a significant number of apps in Android, each application needs to have different access to the device resources in order to prevent security and privacy hazards. To this end, each Android app is a different OS user, to which the user, owner of the device, grants different permissions. By isolating each app, Android guarantees the integrity of the contents of each procedure and prevents other apps from accessing them. Moreover, since the user selects which apps are allowed to access specific resources, the user is able to control the information flow in his device. The latter was significantly improved in Android Marshmallow, as Google decided to introduce the runtime-permission model so that users can grant and revoke app permissions on “dangerous” resources, the ones that present the biggest privacy risk, for instance camera, microphone, and location. See Figure 2 for the full list of the so-called “dangerous permissions”.

To enforce the permission model Android has to perform several steps. Before describing this specific mechanism, it has to be highlighted that since each application in Android is considered as a different user, it is assigned a different UID. This prevents applications from accessing the data and private resources of the other installed apps, providing more security and privacy to Android. Each call to Android framework API is accompanied by the corresponding UID

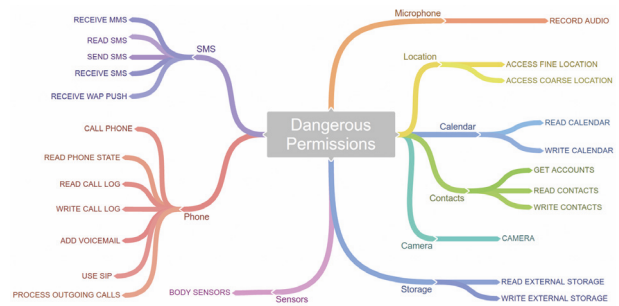


FIGURE 2: Dangerous permissions in Android [5].

of the app performing the call. Android checks whether the permission for the call has been assigned upon installation in the AndroidManifest.xml file and if this is the case, Android checks the permission level of this call (normal, dangerous, etc.). Normal permissions are automatically granted and access to the API is provided instantly. However, if the call is for a dangerous permission, the system will query whether access to this resource has been granted by the user during runtime and allow or deny the access accordingly. Finally, if the permission is signature or signatureOrSystem, then it is granted only to applications that are in the Android system image or that are signed with the same certificate as the application that declared the permission.

While this model may seem secure it does not prevent privacy exposure. The fact that apps have unrestricted access to the Internet allows them to communicate a lot of information. The latter is augmented by the fact that apps can profile their users with normal permissions as they may know, e.g., the apps that are installed, the WiFi networks a user has stored and is using, created and joined arbitrary networks, or even the users' whereabouts with features like WiFi P2P [6]. In general, while an app may use only normal permissions, this does not necessarily mean that it is benign [4, 5, 7]. An adversary model for exfiltrating data from Android devices has been studied in [8] as the use cases are numerous especially in an era when phones are shipped with numerous bloatware [9, 10].

SystemOfSignature permission allows Android apps to be granted a permission as long as an app with the same signature is granted this permission. Extending this concept, Davi et al. [11] showed that apps could escalate their access privileges by performing calls to other applications which had already been granted the privileges they wanted. Orthacker et al. [12] further extended the aforementioned scenario to show that an adversary could use *permission spreading*, that is, split the necessary privileges to different applications, and launch the attack through intercommunication. Similar approaches with app collusion and spread of permissions have been reported in the literature [13–15]; therefore researchers have been gradually focusing on more thorough analysis of intents [16].

In most of the Android cases documented by researchers, information is leaked from one app to another through a covert channel [17, 18]. Although Rushanan et al. in [19]

achieve a goal similar to ours, their study concerns only the desktop environment. Their approach consists in exploiting the Web Workers API in order to increase the CPU and memory utilization. By monitoring both CPU and memory usage, they manage to pass messages from a web page to an app in a desktop computer. Nevertheless, this attack scenario is not possible in an Android device. For devices up to Marshmallow, while apps could monitor the `/proc/` directory and extract some information about memory usage, the recovered information is far from being considered fine-grained and does not include CPU usage. With the introduction of Nougat, apps are allowed to only access the contents of their own `/proc/PID` private directory (<https://developer.android.com/about/versions/nougat/android-7.0-changes.html>), so this method does not work anymore for AOSP. The only other alternative for an app to have this kind of access is to request the system-level permission `PACKAGE_USAGE_STATS` (<https://developer.android.com/reference/android/app/usage/UsageStatsManager.html>). The fact that their attack does not apply for passing messages in Android is also proved by the authors' statement that in Android they managed just to launch a resource depletion attack against the browsers. Moreover, the aforementioned restrictions in Nougat prevent apps from accessing `/proc/net` which could otherwise reveal the domain names but not the full URL a user has visited.

Notably, developers in many occasions, despite Google's recommendations (<https://developer.android.com/training/articles/user-data-ids.html>), use `ANDROID_ID` as a unique identifier. To restrict this, Google required that apps request the dangerous permission `READ_PHONE_STATE` (https://developer.android.com/reference/android/Manifest.permission.READ_PHONE_STATE). Clearly, since this ID is unique, installed apps may identify instances and correlate users and behaviours. Since such actions violate user privacy, even though they are performed locally only among installed apps, in the latest preview of Android O, Google decided to block this behaviour so that each app receives a different `ANDROID_ID`. More precisely, in Android O for each combination of application package name, signature, user, and device, developers end up with a different `ANDROID_ID` (<https://developer.android.com/preview/behavior-changes.html>). To further support users controlling their unique identifiers, Google has recently announced the new changes coming in Android O [20], regarding device identifiers. In this regard, Android O is limiting the use of device-scoped identifiers that are not resettable and is also updating the way that applications request account information, providing more user-facing control. The latter signifies that Google is not only aware of such deanonymization issues, but also constantly working on refining its platform to mitigate these threats and restrict unauthorized and unregulated app-to-app communication, let alone web-to-app communication.

Finally, as reported in [21], there are alternative approaches to `ANDROID_ID`. These methods include, but are not limited to, application metadata in the installation folders or metadata from the `procfs` file system. Nevertheless, all these IDs are related to apps and cannot be used to create an ID that a web page could normally have access to.

2.2. Ad Networks. The freemium model is currently the default monetization method in both web services and native Android apps. The main concept of this model is that users may obtain a product which comes in the form of a service or an app for free in some exchange from the user, which is not directly monetary. In the initial form, the trade involved the user having to watch specific ads; however, in the current form, the model monetizes the data which are generated by the user by using the app or service or the ones that are collected from the user, directly or indirectly. This approach has led many to question the ethicality of this model as the actual product is the user and not the app or service.

To clarify the issue one needs to understand that by correlating a considerable amount of information about a user a lot of sensitive information, hence valuable, can be extracted. For instance, by usage statistics one can determine the interests and preferences of a user, when the user may need or want a specific product or service and therefore create a very fine-grained profile for him that is generated without his consent nor his knowledge. In turn, the companies that can collect these data may sell them for, e.g., targeted advertising, tailored to the exact profile of their users, drastically increasing their success.

The above have radically changed the app and web industry, making ad networks among the most highly valued and influential sectors in these fields. In terms of Android apps, the most widely used ad library is Google's Admob; nonetheless, apps often use more than one. In many occasions, ad libraries have proven not to be benign and to exploit the permissions that they have. Note that due to inheritance the ad libraries have the same permissions that are granted to the apps. Furthermore, it should be highlighted that since ad networks are the sole monetization method for freemium apps, developers are following the wills and commands of ad networks by constantly requesting more and more permissions from their users to collect even more data from them.

Stevens et al. [22] found that some of them would use undocumented permissions, read/write to calendar or access location and camera. Grace et al. [23] found that about half of them would probe the corresponding apps to determine whether they could abuse them to harvest sensitive user information. Ads may perform WiFi scans to determine users' location, scan whether the user has accounts in social networks, or even scan the device to find which applications have been installed [24]. In a more sinister scenario, ad libraries try to link devices by playing inaudible sounds [25]. All the above have led researchers to introduce methods to detain them and restrict the access of ad networks and their user profiling methods [26–29]. Notwithstanding their invasiveness, to the best of our knowledge, none of them has been able to pass information from a browser to an app within the Android system. Instead, this kind of communication, whenever reported, was strictly among apps that used the same ad network.

2.3. Web Fingerprinting Techniques. One of the initial ways to track users was through browser cookies. While they can

easily be removed, it was shown that they could be recovered using the so-called *respawning* method, either using Adobe Flash [30], or using ETags and the HTML5 localStorage API [31]. Moreover, a passive network observer could use third-party HTTP tracking cookies to identify users [32].

More advanced methods try to exploit specific characteristics of either the browser or the device to determine whether a specific browser or device has visited a web page in the past. Typical examples involve browser profiling through collecting information like user agent, installed plugins, supported fonts, time zone, language, etc. Depending on the build and user configuration, these characteristics can be used to identify a browser. Nonetheless, many of these characteristics may change due to updates or user intervention.

A more sophisticated approach involves *canvas fingerprinting*, introduced by Mowery and Shacham [33]. The basic concept is that depending on the underlying operating system, font library, graphics card, graphics driver, and browser, a text or an image can be rendered differently. An adversary could track these changes to derive a browser fingerprint. In fact, such characteristics could be correlated across browsers of the same device as the many WebGL characteristics that can be extracted from each browser offer enough entropy for anonymization [34].

However, such variations can also be traced via other methods. More precisely, regarding smartphones it has been shown that hardware components such as accelerometers, speakers, and microphones may have unique characteristics which differ not only from model to model, but also across devices [35–37]. Obviously, since these characteristics are hardware based, they cannot change and are therefore more robust than typical browser specific methods, realizing the concept of *device fingerprinting*.

3. Session Fingerprinting

When someone browses the Internet, his session is considered anonymous unless he has authenticated himself by logging-in, in a web page. While this anonymity is very convenient for ensuring users' privacy, companies strive to find ways to bypass it and to profile them. Frequently, the benign goal behind these actions is usually regarded as the adaptation and/or personalization of a web page according to the corresponding user profile, which translates to better usability and increased content quality, which in turn may increase both views and viewers. In most of the cases, this personalization also targets the advertising industry, since by deanonymizing an individual a business is able to display ads tailored to users' preferences and therefore to increase both business' and service providers' profits.

One of the most widely used methods in achieving this is browser fingerprinting, which tries to deanonymize users by exploiting noticeable differences in the usage of different browsers such as the underlying OS, user agent, browser version, monitor size, or even installed fonts and plugins [33, 38, 39]. More advanced methods go a step further by exploiting device specific variations to identify individual devices. For smartphones, it has been shown that sensors,

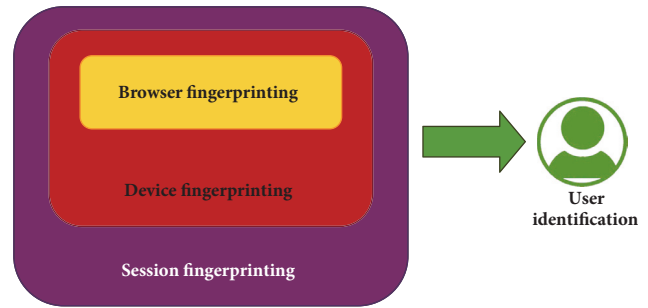


FIGURE 3: Session fingerprinting compared to other methods in the literature.

such as accelerometers, or speakers and microphones may have unique characteristics which differ, not only across models, but also across the devices within which they operate due to calibration errors and frequency distortions [35–37].

While these methods have been proven efficient in many cases, they are usually subject to errors and software updates which could render a previous fingerprint useless. For instance, a browser update may change the user agent or the fonts, making its linking to the previous fingerprint impossible. However, what a company actually needs is to be able to correlate information with other affiliated parties in order to determine whether the user has been simultaneously operating another *session*. A typical example can be regarded as the parallel usage of a web page and a mobile app. Note that while the latter implies that the app is running in the background, it is a typical situation in almost all mobile OSes. In this regard, both parties should try to create a unique ID for each session and also communicate it with each other in order to deanonymize the user. We name the methods for extracting these IDs as *session fingerprinting*. Figure 3 illustrates the relation of our methods to others in the literature. In principle, these methods identify a device; therefore they include device fingerprinting methods; however, since they also include unique identifiers that are collected at each session that a user has with a web page, they amplify the former.

Apparently, these methods depend on the existence of cooperating apps in the mobile device. We argue that this is a weak assumption as the considered adversary in this work is mainly an ad network. Due to the prevalence of the freemium model in Android, most applications are free and, most of the times, they come with at least one preinstalled ad component. However, our requirements do not imply escalated privileges; hence the resulting applications are easier to be accepted by the users.

Although both browsers and the Android OS have such privileges, namely, are able to deanonymize the users and have data about them coming from multiple sources, the level of trust a user has to both of them is the highest possible. Users have chosen them because they trust that they will act honestly and they will protect them from threats and, above all, they will not stalk them. Moreover, it is important to highlight that despite the OSes restrictions, the different existing ad frameworks may indeed perform user profiling,

TABLE 1: Capabilities of apps and web pages.

	Native Apps	Instant Apps	Web Pages
Access Device Identifiers	✓		
Device External Storage	✓		
Push Notifications	✓		
List of Installed Apps	✓		
Access Body Sensors	✓		
Direct Communication with Installed Apps	✓		
Receiving Broadcasts from OS or 3rd party Apps	✓		
Run on the Background	✓		
Change Device Settings	✓		
Access User Calendar	✓	✓	
Access motion sensors	✓	✓	
Access Contacts	✓	✓	
Access Phone Calls	✓	✓	
Access Sensors	✓	✓	
Access environmental sensors	✓	✓	
Access Location	✓	✓	✓
Access Microphone	✓	✓	✓
Access position sensors	✓	✓	✓
Access Camera	✓	✓	✓
Access Internal Storage (own storage)	✓	✓	✓
High precision timestamps	✓	✓	✓

but still they cannot escape the browser or the app ecosystem within which they operate.

4. Problem Setting

The introduction of WWW is one of the milestones in modern computing, enabling users all over the globe to collect and share information with their peers. In fact, the emergence of Social Networks and Media has further reshaped the landscape. In principle, the information that can be collected from a web page is derived solely via the launched browser and is strictly limited to the browser's environment. Any attempt to access resources beyond the browser sandbox is considered as a security violation and therefore is characterized as malicious. To this end, browsers allow very limited exposure of user data to a web page. For instance, a web page cannot read from or write to the storage of a mobile device unless this action is user initiated. To overcome these restrictions, adversaries may resort to browser extensions [40] which provide even more capabilities. On top of that, nowadays, due to the growth of mobile devices, several standards, like HTML5, have been introduced to allow browsers to access additional resources, such as location, camera, or microphone, upon direct and explicit user consent. As a result, web pages have a growing set of capabilities, yet quite limited in comparison to apps. It is worth noting, however, that Chrome, the default Android Browser, as well as many other browsers do not support extensions in the mobile environment, even though they have numerous desktop editions. Finally, up to recently, Chrome apps, also available only for desktop installations,

are discontinued as of early 2018. Therefore, browsers in mobile environments have significantly less functionality and extendability than their desktop peers.

On the other hand, Android apps reside on a different environment. Contrary to web pages, mobile apps “live” directly in the operating system and thus have more direct access to its hardware and corresponding resources. Again, their access is limited according to the granted privileges by the users plus their scope is more fixed as they fulfill specific user needs. Due to this restriction, apps cannot determine which web pages a user visits. A critical distinction between Android apps and web pages is that apps always pass through an “installation” process. This step represents a user acknowledgment regarding the specific resources an app is allowed to use inside the environment executed. Notably, Android Marshmallow users may grant and revoke permissions to specific resources, like camera, microphone, location, etc., which are called “dangerous” and may hinder security and privacy issues. On the contrary, this is not the case for web pages where users are not faced with preconditions for visiting them. In many instances, users would like to be able to use “one-time apps” to accomplish specific tasks like using a retailer’s app when browsing his web page. To address this need, Google recently introduced *instant apps*, which do not require installation and have more permissions and/or capabilities than common web pages. However, instant apps, like web pages, are also restricted from accessing hardware identifiers to prevent user profiling.

Key differences in terms of the capabilities of Android apps, native and instant, and web pages are illustrated in Table 1. As expected from the earlier discussion, it can

be easily noticed that installed applications have far more access to device resources than web pages, since users install apps granting themselves the corresponding permissions. On the contrary, due to the nature of the web, users may visit a large number of different web pages on a daily basis, without knowing their quality, intentions, source, or content. Hence, both Android and browsers make significant efforts, e.g., running in a sandbox environment, towards protecting users from malicious web page behaviour. Unquestionably, if an app had been able to communicate with a web page without restrictions, the entire underlying security infrastructure would have been rendered useless. In fact, even the most widely used apps in Android are not able to communicate directly with their web page. For instance, in the case of three well-known and widely used apps, Facebook, Instagram, and Twitter, they do not transfer information to their corresponding web page or any other cooperating web page when a user has logged in the app. Instead, a “Connect with Facebook/Twitter” button usually appears, requiring further user interaction and most importantly being realized by users. Evidently, had these apps been able to transfer this kind of information to the browser, they would have done it already long time ago, not only for facilitating users, but also for further increasing the amount of collected user data and the quality of provided services.

Creating a mechanism being able to transmit an identifier from the browser to a cooperating installed app in a user’s device, or vice versa, would allow for the installed app to identify the individual who visited the cooperating web page and subsequently the web page would elevate its access to the same resources as those of the installed app. Further analyzing this, after both parties, namely, web pages and apps, have identified themselves lying in the same user’s device, they would be able to create a covert channel. In the least sinister scenario, a web page cooperating with an app would be able to access a user’s contacts, SMS messages, or even storage and microphone, without obtaining user’s consent, and would manage to display ads perfectly tailored to the user’s profile, albeit violating his privacy. However, in a true malicious scenario, user data would be harvested by web pages and personalized exploits would be pushed to users’ devices to further exploit their personal data while they surf in the WWW.

5. Intercommunication between Apps and Web Pages

In the following subsections, we provide a set of concrete examples as Proofs of Concept, which showcase how apps and web pages can mutually create and consequently transmit unique IDs that allow them to link their usage and to communicate sensitive attributes to each other, realizing what the authors of this paper have introduced as “session fingerprinting”. The authors of this paper have responsibly disclosed the mentioned security issues described in the next subsections, regarding unauthorized communications between apps and web pages.

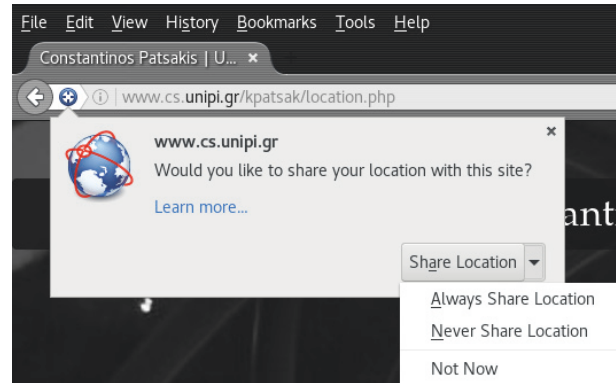


FIGURE 4: Firefox requests user’s permission to allow a web page to access location.

5.1. Location. Location awareness has undoubtedly increased the potential of many applications since it allows them to adapt accordingly and render their information based on location specific criteria, drastically improving, e.g., user recommendations. Obviously, location is a sensitive piece of information as it can disclose many private attributes, ranging from work and residence location, to entertainment preferences and political/religious beliefs if correlated with other sources of information. Therefore, mobile OSes allow applications to access location data only if the user grants a corresponding permission. In Android this permission is provided either as *fine* or as *coarse* location.

Similarly, beyond the support for media in HTML5, the standard enables web pages to access user location. Since this information is sensitive, the browser specifically requests user permission to be granted—see Figure 4—even though this kind of information can be used for other purposes as well. Once the browser gets access to the user’s location, the response contains apart from the longitude and the latitude the accuracy and the timestamp [41] as well. Moreover, depending on the implementation, it may also return other values, such as heading and speed. Interestingly, in our research we have come up with proofs that this information can be correlated with location data information from an Android app. More precisely, while an Android app could monitor a user’s location and is able to correlate the coordinates with the ones that are received from a web page, one could argue that since these requests to location data are not made simultaneously, from the browser and the app, the actual identity of the user is not disclosed. This argument can be clearly supported either because other nearby users may be also implicated, or because the web page gets this information only once. However, practically this is not the case. For reasons such as minimizing battery consumption, since the usage of the GPS is rather greedy, Android app developers may choose to use the “last known location” feature through the `getLastLocation()` method of `LocationServices` which fetches the location from its cache [42]. Based then on the accompanied timestamp the developer can determine whether he needs to request a new reading or not. Yet, what seems quite interesting in this case is that our findings reveal

that, by accessing the device's last known coarse location from an app, we actually end up having data about the precise last location request made by a web page. It should be emphasized that "coarse" location is the one that should be used here, since "fine" location is accessed exclusively by Android apps and not web pages and hence is not suitable for our method.

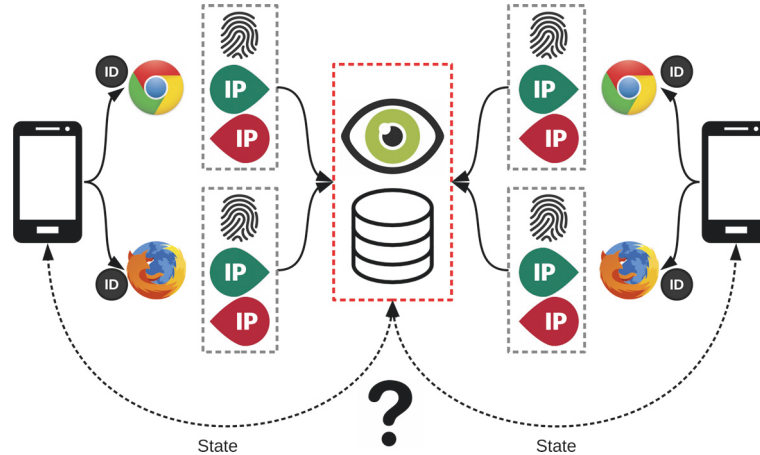
Apparently, if a mobile app monitors the last known location and its corresponding timestamp determined by the Network Location Provider and communicates this information to the "All Seeing Eye", the latter is able to determine whether it coincides with user's coordinates and timestamp received from a web page. Beyond a doubt, this combination of data is quite "unique". Once a correlation is found, the All Seeing Eye can create a covert channel that will serve both the app and the web page to exchange data regarding this session. This specific instance of "session fingerprinting" has been successfully tested in all recent Android versions, from Marshmallow to Oreo. Nevertheless, it is expected to be fully functional to all Android versions, since it utilizes one of the most basic and native Android mechanisms, namely, location services, available since API level 1. Geolocation W3C API of HTML5 has been also been available since the first versions of mobile browsers, namely, Android Browser, Samsung Internet, and Google Chrome [43].

5.2. Browser Fingerprinting. In the previous example a dangerous, yet very commonly used permission, namely, location, was used to identify a user. Nevertheless, one could achieve the same result without such permissions. A more stealth method is to utilize browser fingerprinting. To this end, we assume that the victim has installed an application which does not request any dangerous permission. According to the Android permission model, such applications are allowed to list all the installed applications in a device; hence the adversary has also knowledge of all the installed browser applications. This way, the app can subsequently open all the available browsers through intents and point them to a desired URL in order to obtain a fingerprint from them. Note that, as for Nougat, an application cannot determine which is the foreground application, a piece of information that would have been very valuable for the adversary; however, the authors of this paper have already notified Google of a new method to achieve this in all versions prior to Nougat (Android Issue no 23504, triaged). Each time a browser is fingerprinted by the app, a random nonce is created and is sent in the web page request allowing the adversary to determine to whom its fingerprint belongs. This kind of attack utilizes malicious intents, while for "covering traces" purposes the cooperating malicious web pages could be redirected to a commonly used web page (e.g., a search engine), after accomplishing the ID exchanging job. A scenario where no intents are needed also exists, where a malicious app may use its native webview component in order to accomplish the aforementioned task.

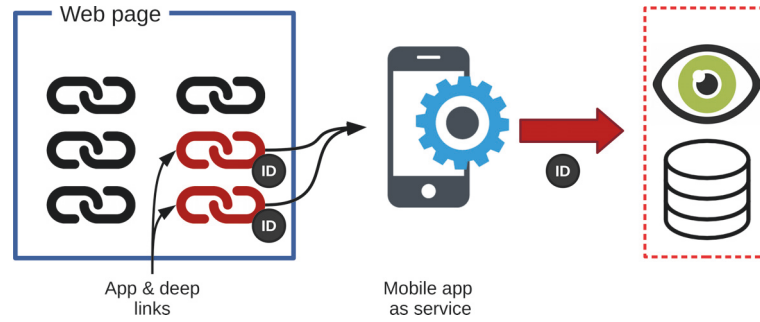
Since mobile devices have less "unique" characteristics compared to personal computers, an extension to browser fingerprints is to additionally use even more mobile device

characteristics, further conforming to the "session fingerprinting" proposed term. Indeed, both mobile app and web page can obtain knowledge about the internal and the external IP of the mobile device. For the former one could potentially use WebRTC [44] which is known to leak several pieces of private information [45]. Therefore, when a user visits a web page, the web page queries the "All Seeing Eye" to determine if someone with the specific browser fingerprint and public and local IPs has a cooperating app running at a specific timeframe. In general, the chances of this query returning more than one result are slim. Nonetheless, in a corporate environment, where many people might have mobile devices of the same model, some instances may exist. In such environments one could have two identical devices with the same internal IP, if, e.g., two users with the same smartphone model use the corporate WiFi on different floors or departments. To further reduce the query results, the "All Seeing Eye" could request the state of each device. In this case, additional information that can be cross-checked between apps and web pages includes, but is not limited to, the following: battery information (both state and charging level), interval since device's last noticeable movement (this could be determined, e.g., via accelerometers), interval since last proximity (via proximity sensor), light measurements, positioning (e.g., facing up or down), or even some connection statistics such as `downlinkMax` (one of the new features of HTML5 through the Network Information API [46]). From this information one can easily determine which user is using a smartphone at a specific timeframe and essentially eliminate the possibilities of having false positives. This process is illustrated in Figure 5(a). In this figure, we may notice that both web pages inside browsers and also web pages inside applications' webview components are able to collect unique fingerprints, namely, internal and external IPs, accompanied by information regarding the devices' state and communicate them to the "All Seeing Eye" which will then be responsible for finding exact matches between them. Due to the huge amount of data that have to be correlated, approaches like [47] can be used to boost the performance. HTML5s WebRTC W3C API is available to all Android smartphones running Samsung Internet browser and Google Chrome. Network Information API is available on Android smartphones with Android Internet browser version ≥ 2.2 and Google Chrome version ≥ 38 , [43].

5.3. App and Deep Links. Most users install plenty of apps on their devices, even though many of them might have some overlapping functionality. To facilitate user interaction between websites and Android applications the Web Intents framework was introduced, allowing a developer to specify how a hyperlink is handled on the user device, e.g., open the phone to dial up an already prepared number or use Skype for a specific contact. However, Android supports further features through *app* and *deep links*. The concept behind both of these types of web links is to open specific apps depending on the link. As an example, Facebook and Twitter apps are triggered when the user taps on a link referring to content of the corresponding site.



(a) Identification through browser fingerprinting



(b) Identification through app and deep links

FIGURE 5: User identification methods.

Interestingly, this kind of functionality is automatically activated when a user installs an application which provides such features, without requesting any user approval. Moreover, Android activities may run on the foreground in a “hidden” mode, either by using transparent themes or by utilizing floating zero-sized activities. Practically, this creates a hidden communication channel between web pages and apps that can be used to identify users as illustrated in Figure 5(b). We assume that an app is installed in a user’s device having at least one “browsable” (declared inside Apps Manifest file) activity in order to enable “cooperation” with web pages. On the other side, web pages embed some special “intent” hyperlinks which also have the ability to carry a random ID, different for every interaction. Once a user taps in one of these links, the ID is bundled and transferred to the app, using the “getExtras” method inside the “Intent” Android class. As a final step, once again the data is communicated to the All Seeing Eye, deanonymizing the user. This kind of “interaction” can be seen as a directed web-to-app communication, where an app has the ability to be reached from web pages. At the same time one or more web pages may utilize this “functionality” by transmitting an ID which is essential for the entire described scenario. Next subsection describes how this local channel communication can be achieved through the opposite direction of interaction. Regarding “browsable” activities that enable deep linking, this

feature is available in Android since API level 1. As a result, this instance of “session fingerprinting” is also expected to affect all versions of Android.

5.4. Direct App to Web Communication. Following the same logic as in the previous subsections, an app is also able to directly reach a web page through a device’s installed browser. Once again, Android Intents are deployed in order to launch an installed browser and to pass a specific URL. The cooperating app is able to inject one or more string values inside the URL as parameters, which in our case is a simple, randomly generated ID, and correspondingly fire a malicious intent towards a browser. As a next step, the loaded web page can extract the ID from the URL’s “location search” property and thus a local covert channel between the app and the launched web page is realized. Naturally, the web page is again able to communicate the ID to the “All Seeing Eye”, making the user’s profile available to others as well. As already discussed, this kind of method “leaves some traces”; namely, it opens a web browser; however the corresponding malicious web page can hide its traces with a simple redirection, e.g., to a search engine’s default page.

A quite significant detail in this process is that the corresponding web page should initially use a client side programming language to retrieve the transmitted ID. This is essential in order to create the local covert channel between

TABLE 2: Privacy exposure to the described threats per API level.

Method	API level
Location	> 1
Browser fingerprinting	Browser specific. Native WebView > 20
App and deep links	> 1
Direct App to Web communication	> 1

TABLE 3: Necessary permissions for each Android ad network.

	% of installing	Internet	ACCESS_NETWORK_STATE	WRITE_EXTERNAL_STORAGE	ACCESS_WIFI_STATE
Admob	61.73	✓			
Unity Ads	19.02	✓	✓		
Chartboost	14.07	✓	✓		
MoPub	13.62	✓	✓		
AdColony	13.18	✓	✓		
AppLovin	12.91	✓		✓	
AppsFlyer	10.23	✓	✓	✓	✓
InMobi	9.14	✓	✓		
Vungle	7.08	✓	✓	✓	
Tapjoy	6.72	✓	✓		✓

TABLE 4: Results from Tacyt.

	Google Play		Other markets	
	Available	Unavailable	Available	Unavailable
App & Deep links	432,204	87,483	71,686	34
ACCESS_COARSE_LOCATION	996,326	215,335	154,749	29
INTERNET	4,044,922	1,046,310	533,492	149
Total versions in market	4,207,542	1,095,398	576,204	155

the app and the web page, since an app ID directly transmitted to a web server would lose the ability to “find its way back” to the corresponding device’s web page. This fourth instance of “session fingerprinting” also utilizes native Android mechanisms, since even the first Android smartphones were able to use intents and communicate with web pages through browsers. As a result, it has been successfully tested on all recent Android versions, from Marshmallow to Oreo.

6. Experimental Results and Statistics

In Table 2 we illustrate from which API levels Android devices are exposed to the threats we have described, indicating that these methods apply to the vast majority of devices available.

In order to provide an estimation of the potential exposure of users to these threats, the authors of this paper used data by utilizing “Tacyt” (<https://tacyt.elevenpaths.com>). Tacyt is an innovative cyber intelligence tool that facilitates research in Android mobile apps environments with big data technology. The aim of Tacyt is to enable quick detection, discovery, and analysis of these threats in order to reduce their potential impact on organizations. Enabling app data mining and detection enables research and analysis of the collected information from Google Play and other markets. Due to the

implementation of Tacyt, the query responses are per app version and not per app; nonetheless they provide a very good overview of apps dating back to at least three years ago.

Table 4 presents the results from the performed queries. In our first query we tried to identify how many apps provide a deep or app link. This information is declared in the manifest of each application and is clearly marked with the `android.intent.category.BROWSABLE` tag of the XML file. The next two rows involve app versions which required the `ACCESS_COARSE_LOCATION` and Internet permission which could be potentially used to deanonymize users. Similarly, using PublicWWW, a source code search engine for web pages, we found more than 96,000 web pages to use geolocation features in their code for locating users. A sample of the attributes received from both APIs is illustrated in Box 1.

Regarding ad networks in Android, we tried to highlight the requirements, in terms of permissions, that each of them request from the developers and whether they could exploit our methods. Analyzing the requested permissions of the top 10 ad networks in Android according to Appbrain (<https://www.appbrain.com/stats/libraries/ad>) (see Table 3 for the corresponding list), we found that all of them required the obvious normal permission of `Internet`. Moreover,

```

float getAccuracy(): Get the estimated horizontal accuracy of this location, radial, in meters.
double getAltitude(): Get the altitude if available, in meters above the WGS 84 reference ellipsoid.
float getBearing(): Get the bearing, in degrees.
float getBearingAccuracyDegrees(): Get the estimated bearing accuracy of this location, in degrees.
long getElapsedRealtimeNanos(): Return the time of this fix, in elapsed real-time since system boot.
Bundle getExtras(): Returns additional provider-specific information about the location fix as a Bundle.
double getLatitude(): Get the latitude, in degrees.
double getLongitude(): Get the longitude, in degrees.
String getProvider(): Returns the name of the provider that generated this fix.
float getSpeed(): Get the speed if it is available, in meters/second over ground.
float getSpeedAccuracyMetersPerSecond(): Get the estimated speed accuracy of this location, in meters per second.
long getTime(): Return the UTC time of this fix, in milliseconds since January 1, 1970.
float getVerticalAccuracyMeters(): Get the estimated vertical accuracy of this location, in meters.
boolean hasAccuracy(): True if this location has a horizontal accuracy.
boolean hasAltitude(): True if this location has an altitude.
boolean hasBearing(): True if this location has a bearing.
boolean hasBearingAccuracy(): True if this location has a bearing accuracy.
boolean hasSpeed(): True if this location has a speed.
boolean hasSpeedAccuracy(): True if this location has a speed accuracy.
boolean hasVerticalAccuracy(): True if this location has a vertical accuracy.
boolean isFromMockProvider(): Returns true if the Location came from a mock provider.

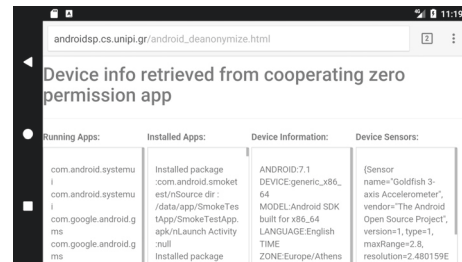
Position.coords: Returns a Coordinates object defining the current location.
Position.timestamp: Returns a DOMTimeStamp representing the time at which the location was retrieved.
Where coordinates contain the following information:
double Coordinates.latitude: The position's latitude in decimal degrees.
double Coordinates.longitude: The position's longitude in decimal degrees.
double Coordinates.altitude: The position's altitude in meters, relative to sea level. Can be null
if the implementation cannot provide the data.
double Coordinates.accuracy: he accuracy of the latitude and longitude properties, expressed in meters.
double Coordinates.altitudeAccuracy: he accuracy of the altitude expressed in meters. This value can be null.
double Coordinates.heading: he direction in which the device is traveling in degrees.
double Coordinates.speed: he velocity of the device in meters per second. This value can be null.

```

Box 1: Sample attributes from native Android and JavaScript geolocation APIs.

most of them (8/10) requested the `ACCESS_NETWORK_STATE` permission. Notably, 3 of them requested access to dangerous permissions, while all of them requested writing data in the device (`WRITE_EXTERNAL_STORAGE`). Finally, all of them also proposed the use of location (`ACCESS_COARSE_LOCATION/FINE_COARSE_LOCATION`).

Combining the evidence from the sections regarding session fingerprinting and this section's experimental data regarding ad network permissions, one may easily infer that our proposed techniques for user deanonymization are realistic in terms of permission requirements, since they require either zero or normal permissions or, in some cases, dangerous permissions that are frequently requested by ad networks. What is more alarming is the concern whether this paper sayings are already applied and utilized by existing ad networks, third-party apps, and corresponding web pages. Interestingly, after analyzing the above information we may discuss that imposing restrictions or even applying control mechanisms for the Internet could result in a large benefit towards the users' privacy. Nevertheless, this special, "normal", permission seems more tightly linked to all kinds of advertising and marking it as "dangerous" would require more than strong will by the companies involved.



Running Apps:	Installed Apps:	Device Information:	Device Sensors:
com.android.systemu com.android.systemu com.google.android.g ms com.google.android.g ms	Installed package com.android.smoket est\nSource dir: /data/app/SmokeTes App/SmokeTestApp. apk\nLaunch Activity null Installed package	ANDROID:7.1 DEVICE:generic_x86_ 64 MODEL:Android SDK built for x86_64 LANGUAGE:English TIME ZONE:Europe/Athens	(Sensor name="Goldfish 3- axis Accelerometer", vendor="The Android Open Source Project", version=1, type=1, maxRange=2, resolution=2.480159E

FIGURE 6: Device info as obtained from a web page through a zero-permission app installed in a device running Android 7.1.1.

Finally, we have implemented a proof of concept app that is able to cooperate with web pages without requesting any dangerous permissions. The app is available at androidsp.cs.unipi.gr/android_deanonymize.html. Once installed, the app recovers a lot of information from the device, such as installed and running apps and device info as well as measurements from many sensors which do not require any dangerous permissions. Eventually, as illustrated in Figure 6, when the user visits the corresponding web

page through his mobile browser he can verify that the web page has recovered all this information without requesting his consent. It is worth noting that, apart from providing access to sensors that a web page would not normally have, the measurements for these sensors are also listed in a fine-grained mode, e.g., access to accelerometer detailed measurements which could be used for fingerprinting [35].

7. Conclusions

The ever increasing use of mobile devices exposes user privacy in numerous ways. Despite the fact that mobile OSes take several measures to protect their users, attackers seem to always be one step ahead. Nonetheless, most would agree that the state-of-the-art countermeasures guarantee an independence between the browser and the mobile apps so they cannot exchange information. Taking Android as our reference platform, we introduce new methods that exploit various inherent mechanisms to practically guarantee absolute identification with limited resource usage. Moreover, the proposed methods extend the notion of device fingerprinting to what we have introduced as “session fingerprinting”. Our techniques can be performed without accessing unique device characteristics or using dangerous permissions. In this regard, our techniques imply a bigger threat, as the covert channel that is created between the web pages and the apps cannot be easily traced.

Due to the fact that all the aforementioned mechanisms are inherent in Android, one cannot rule out the possibility that these mechanisms are already being exploited, enabling unauthorized and unregulated cooperation between the two ecosystems. Clearly, this would greatly expose users’ privacy, bypassing the permission model of the most widely used mobile platform to date. Addressing such issues is a rather challenging task, because, apart from changing the native Android mechanisms, it is also a requirement for the OS to determine the context of some function calls, either to prohibit access to resources or to obfuscate the underlying information, since these calls might seem legitimate.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the OPERANDO project (Grant Agreement no. 653704). The authors would like to acknowledge *ElevenPaths* for their valuable feedback and providing them access to Tacyt.

References

- [1] The Economist, “The world’s most valuable resource is no longer oil, but data,” 2017, <https://www.economist.com/news/leaders/21721656-data-economy-demands-new-approach-anti-trust-rules-worlds-most-valuable-resource/>.
- [2] E. Alepis and C. Patsakis, “The All Seeing Eye: Web to App Intercommunication for Session Fingerprinting in Android,” in *Security, Privacy, and Anonymity in Computation, Communication, and Storage*, vol. 10656 of *Lecture Notes in Computer Science*, pp. 93–107, Springer International Publishing, 2017.
- [3] Or. Peles and Roe. Hay, “One class to rule them all: 0-day deserialization vulnerabilities in android,” in *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT 15)*, USENIX Association, 2015.
- [4] E. Alepis and C. Patsakis, “Trapped by the UI: The Android Case,” in *Research in Attacks, Intrusions, and Defenses*, vol. 10453 of *Lecture Notes in Computer Science*, pp. 334–354, Springer International Publishing, 2017.
- [5] E. Alepis and C. Patsakis, “Hey Doc, Is This Normal?: Exploring Android Permissions in the Post Marshmallow Era,” in *Security, Privacy, and Applied Cryptography Engineering*, vol. 10662 of *Lecture Notes in Computer Science*, pp. 53–73, Springer International Publishing, 2017.
- [6] E. Alepis and C. Patsakis, “There’s Wally! Location Tracking in Android without Permissions,” in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, pp. 278–284, Porto, Portugal, February 2017.
- [7] E. Alepis and C. Patsakis, “Monkey Says, Monkey Does: Security and Privacy on Voice Assistants,” *IEEE Access*, vol. 5, pp. 17841–17851, 2017.
- [8] Q. Do, B. Martini, and K.-K. R. Choo, “Exfiltrating data from Android devices,” *Computers & Security*, vol. 48, pp. 74–91, 2015.
- [9] P. McDaniel, “Bloatware comes to the smartphone,” *IEEE Security & Privacy*, vol. 10, no. 4, pp. 85–87, 2012.
- [10] H. Elahi, G. Wang, and L. Xu, “Smartphone bloatware: An overlooked privacy problem,” in *Proceeding of the Security, Privacy, and Anonymity in Computation, Communication, and Storage - 10th International Conference, (SpaCCS ’17)*, G. Wang, M. Atiquzzaman, Z. Yan, and K. K. R. Choo, Eds., vol. 10656 of *Lecture Notes in Computer Science*, pp. 169–185, Guangzhou, China, 2017.
- [11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in *Information Security*, vol. 6531, pp. 346–360, Springer, 2011.
- [12] C. Orthacker, P. Teufl, S. Kraxberger et al., “Android security permissions—can we trust them?” in *Security and Privacy in Mobile Information and Communication Systems*, pp. 40–51, Springer, 2012.
- [13] A. Dimitriadis, P. S. Efraimidis, and V. Katos, “Malevolent app pairs: an android permission overpassing scheme,” in *Proceedings of the ACM International Conference on Computing Frontiers, CF ’2016*, pp. 431–436, ACM, New York, NY, USA, May 2016.
- [14] F. Vincent Taylor, R. Alastair Beresford, and Ivan. Martinovic, “Intra-library collusion: A potential privacy nightmare on smartphones,” *arXiv preprint arXiv:1708.03520*, 2017.
- [15] J. Blasco and T. M. Chen, “Automated generation of colluding apps for experimental research,” *Journal of Computer Virology and Hacking Techniques*, pp. 1–12, 2017.
- [16] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, “AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection,” *Computers & Security*, vol. 65, pp. 121–134, 2017.
- [17] W. Gasior and L. Yang, “Exploring covert channel in android platform,” in *Proceedings of the ASE International Conference on Cyber Security (CyberSecurity ’12)*, pp. 173–177, Washington, DC, USA, December 2012.

- [18] S. Chandra, Z. Lin, A. Kundu, and L. Khan, "Towards a Systematic Study of the Covert Channel Attacks in Smartphones," in *International Conference on Security and Privacy in Communication Networks*, vol. 152 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 427–435, Springer International Publishing, 2015.
- [19] M. Rushanan, D. Russell, and A. D. Rubin, "MalloryWorker: Stealthy Computation and Covert Channels Using Web Workers," in *Security and Trust Management*, vol. 9871 of *Lecture Notes in Computer Science*, pp. 196–211, Springer International Publishing, 2016.
- [20] Android Developers Blog, "Changes to device identifiers in android o," 2017, <https://android-developers.googleblog.com/2017/04/changes-to-device-identifiers-in.html>.
- [21] E. Alepis and C. Patsakis, "Persistent vs Service IDs in Android: Session Fingerprinting from Apps" in *Mobile Networks and Management*, vol. 235 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 14–29, Springer International Publishing, 2018.
- [22] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Proceedings of the 2012 Workshop on Mobile Security Technologies*, 2012.
- [23] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec'12*, pp. 101–112, USA, April 2012.
- [24] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," *arXiv preprint arXiv:1303.0857*, 2013, <https://arxiv.org/abs/1803.03270>.
- [25] D. Goodin, "Beware of ads that use inaudible sound to link your phone, tv, tablet, and pc," 2015w <http://arstechnica.com/tech-policy/2015/11/beware-of-ads-that-use-inaudible-sound-to-link-your-phone-tv-tablet-and-pc/>.
- [26] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: privilege separation for applications and advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*, pp. 71–72, Seoul, Republic of Korea, May 2012.
- [27] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *Proceeding of the Presented as part of the 21st USENIX Security Symposium (USENIX Security'12)*, pp. 553–567, 2012.
- [28] X. Zhang, A. Ahlawat, and W. Du, "AFrame: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference, (ACSAC '13)*, pp. 9–18, December 2013.
- [29] V. Tsiakos and C. Patsakis, "AndroPatchApp: Taming Rogue Ads in Android," in *Mobile, Secure, and Programmable Networking*, vol. 10026 of *Lecture Notes in Computer Science*, pp. 183–196, Springer International Publishing, 2016.
- [30] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle, "Flash cookies and privacy," in *Proceedings of the 2010 AAAI Spring Symposium*, pp. 158–163, March 2010.
- [31] M. Ayenson, D. J. Wambach, A. Soltani, N. Good, and C. J. Hoofnagle, "Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning," *SSRN Electronic Journal*, 2011.
- [32] S. Englehardt, D. Reisman, C. Eubank et al., "Cookies that give you away: The surveillance implications of web tracking," in *Proceedings of the 24th International Conference on World Wide Web, (WWW '15)*, pp. 289–299, May 2015.
- [33] K. Mowery, S. Keelveedhi, and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," in *Proceedings of W2SP*, p. 19, Raleigh, North Carolina, USA, October 2012.
- [34] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via os and hardware level features," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS*, San Diego, CA.
- [35] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi, "AccelPrint: imperfections of accelerometers make smartphones trackable," in *Proceedings of the Network and Distributed System Security Symposium (NDSS '14)*, San Diego, Calif, USA, February 2014.
- [36] Z. Zhou, W. Diao, X. Liu, and K. Zhang, "Acoustic fingerprinting revisited: Generate stable device ID stealthily with inaudible sound," in *Proceedings of the 21st ACM Conference on Computer and Communications Security, (CCS '14)*, pp. 429–440, USA, November 2014.
- [37] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh, "Mobile device identification via sensor fingerprinting," *arXiv preprint arXiv:1408.1416*, 2014, <https://arxiv.org/abs/1408.1416?context=cs>.
- [38] P. Eckersley, "How unique is your web browser?" in *International Symposium on Privacy Enhancing Technologies Symposium*, vol. 6205 of *Lecture Notes in Computer Science*, pp. 1–18, Springer, Berlin, Germany, 2010.
- [39] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," in *Proceedings of W2SP*, vol. 2, pp. 180–193, 2011.
- [40] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *USENIX Security*, pp. 641–654, 2014.
- [41] A. Popescu, "Geolocation api specification 2nd edition," 2016, <https://www.w3.org/TR/geolocation-API/>.
- [42] Android developers, "Getting the last known location," 2017, <https://developer.android.com/training/location/retrieve-current.html>.
- [43] Mobile HTML, "Html5 compatibility on mobile and tablet browsers with testing on real devices," 2018, <http://mobilehtml5.org/>.
- [44] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba, "WebRTC 1.0: Real-time communication between browsers," 2016, <https://www.w3.org/TR/webrtc/>.
- [45] V. Beltran, E. Bertin, and N. Crespi, "User identity for WebRTC services: A matter of trust," *IEEE Internet Computing*, vol. 18, no. 6, pp. 18–25, 2014.
- [46] M. Cáceres, F. J. Moreno, and I. Grigorik, "Network information API," 2017, <http://wicg.github.io/netinfo/>.
- [47] W. Yang, G. Wang, K. R. Choo, and S. Chen, "HEPart: A balanced hypergraph partitioning algorithm for big data applications," *Future Generation Computer Systems*, vol. 83, pp. 250–268, 2018.

