Investigating Fingerprinters and Fingerprinting-alike Behaviour of Android Applications

Christof Ferreira Torres[1,2] and Hugo Jonker[3,4]

[1] Fraunhofer AISEC, Munich, Germany
[2] SnT, University of Luxembourg, Luxembourg, Luxembourg
`christof.torres@uni.lu`
[3] Open University of the Netherlands, Heerlen, Netherlands
`hugo.jonker@ou.nl`
[4] Radboud University, Nijmegen, Netherlands

Abstract. Fingerprinting of browsers has been thoroughly investigated. In contrast, mobile phone applications offer a far wider array of attributes for profiling, yet fingerprinting practices on this platform have hardly received attention.

In this paper, we present the first (to our knowledge) investigation of Android libraries by commercial fingerprinters. Interestingly enough, there is a marked difference with fingerprinting desktop browsers. We did not find evidence of typical fingerprinting techniques such as canvas fingerprinting. Secondly, we searched for behaviour resembling that of commercial fingerprinters. We performed a detailed analysis of six similar libraries. Thirdly, we investigated ∼30,000 apps and found that roughly 19% of these apps is using one of the these libraries. Finally, we checked how often these libraries were used by apps subject to the *Children's Online Privacy Protection Act* (i.e. apps targeted explicitly at children), and found that these libraries were included 21 times.

1 Introduction

Fingerprinting is a side-channel approach to identifying devices. Instead of using an explicitly defined identifier (e.g. an HTTP cookie), fingerprinting relies on determining a set of characteristics that together are uniquely identifying. This can be used for user tracking as well as fraud prevention (e.g. preventing logins from devices with unknown fingerprints). For desktop internet browsers, Eckersley found [3] that even a small set of attribute values such as screen resolution, browser version, and operating system version is typically sufficient to reliably re-identify a browser and, thereby, a user.

In comparison to desktop browsers, smartphone platforms facilitate fingerprinting better. Smartphones possess a large array of sensors (e.g. accelerometer, GPS, etc.), antennas (e.g. WiFi, Bluetooth, GSM, etc.) and internal characteristics (phone number, contact list, installed apps, etc.), which together provide a large fingerprintable surface. The extent to which these data can be accessed without privileges is far greater than on desktop browsers (e.g. JavaScript access to sensor APIs like the Android device motion API). Moreover, unlike desktop browsers where third-party cookies can be used to track users across sites, the Android API does not provide any features for sharing state between apps. Any cross-app tracking is therefore forced to develop its own approach to re-identification, which, in absence of an explicit identification mechanism, must rely on side channels, i.e. fingerprinting. Moreover, currently Android offers several globally unique identifiers (e.g, MAC address, `ANDROID_ID`, advertising ID etc.). However, their use for tracking is reducing as newer versions of Android take privacy measures to precisely prevent this. In other words: unique identifiers seem not to be future-proof. In contrast, fingerprinting is easy to realize with few permissions, can result in a large set of identifying data, serves cross-app and cross-device tracking, and is more future-proof than relying on identifiers. This provides a strong incentive for using fingerprinting in mobile apps. Unlike browser fingerprinting, mobile device fingerprinting in practice has, to the best of our knowledge, received scant attention.

In this paper, we investigate the extent to which the rich fingerprinting opportunities offered by Android smartphones are being taken advantage of. The results are surprising. A large amount of companies is collecting data that could be potentially used for profiling/identification of users (besides unique identifiers). We show evidence that companies are not just collecting identifiers but much more data. These results are uncovered by a scanning tool: *FP-Sherlock*. This is a static scanner, designed to quickly identify potential fingerprinting apps from a large corpus, reducing the search space sufficiently to make manual inspection feasible.

Contributions. We reverse engineer two commercial fingerprinting libraries and analyse and discuss their workings. Based on this, we design *FP-Sherlock*, a static scanner that can find fingerprinting-alike behaviour in APK files. We apply FP-Sherlock to a corpus of ∼30,000 top apps, which identifies a number of libraries that exhibit such behaviour. We reverse engineer the six most similar libraries and discuss their workings and their occurrence rates within our corpus. Finally, we check how many apps in our corpus include such libraries while being age-restricted, i.e., subject to the *Children's Online Privacy Protection Act* (COPPA), which strictly regulates profiling of children.

2 Background and Related Work

Eckersley [3] was the first to investigate browser fingerprinting. He computed the entropy for various browser attributes, and found that about 90% of desktop browsers were unique in his data set (about 470,000 fingerprints). Commercial fingerprinters were first identified by Mayer et al. [7], identifying (amongst others) Iovation as a company that offers browser fingerprinting services to websites. This triggered research aimed at detecting commercial as well as non-commercial fingerprinters. Nikiforakis et al. [9] investigated the workings of three commercial browser fingerprinters. A work by Acar et al. [1] uncovered a new commercial fingerprinter (ThreatMetrix).

Browser fingerprinting has also been investigated on mobile phones. Spooren et al. [12], Hupperich et al. [4], and Laperdrix et al. [6] all investigated browser fingerprinting on mobile devices. While Laperdrix et al. were positive by using canvas fingerprinting [8], the other two studies reported negative results. Remark that none of these studies leverage the far richer attribute surface of mobile devices, and so are ill-suited for investigating real-life data gathering practices in the mobile domain. Hupperich et al. proposed to break away from browser fingerprinting for mobile device fingerprinting. They introduce four classes of attributes (browser, system, hardware and behaviour) and set up an experiment using 45 attributes over these four classes. Their experiments illustrate the feasibility of fingerprinting mobile devices, as well as the strong reliance dependence of their re-identification process on two attributes in particular. Others have also looked beyond browser fingerprinting into mobile device fingerprinting.

Kurtz et al. [5] investigate device fingerprinting for iOS devices. They manually evaluated the iOS SDK to find fingerprintable attributes, identifying 29 attributes as such, including last played songs, list of installed applications, etc. They created an iOS application to fingerprint devices based on these 29 attributes, and found that in their test set of 8,000 different devices, each device had a unique fingerprint. Wu et al. [15] provide a similar study focused on the Android operating system. They construct a fingerprinting mechanism that does not require permissions, relying on 38 attributes. Wu et al. investigate the entropy of these 38 attributes, finding the list of currently installed applications and wallpapers to be the most revealing.

In summary, the many developments that occurred to desktop browser fingerprinting are not directly applicable to mobile fingerprinting. While Wu et al. [15] demonstrated the possibility of fingerprinting users on Android devices, there is insufficient data available on the entropy of attributes to label determine whether a specific set of gathered attribute values could serve as a fingerprint. Remark that Wu et al. only investigated 38 attributes that served their purpose. In contrast, we seek to find out what attributes are being used by actual fingerprinters in practice, and the extent to which other apps engage in similar behaviour. To the best of our knowledge, we are the first to study whether or not mobile fingerprinting occurs in practice.

3 Investigating Known Fingerprinters

Previous works [9,7,1] had identified three companies behind commercial browser fingerprinting: *BlueCava*, *Iovation* and *ThreatMetrix*. We used the ICSI Haystack Panopticon project [14] to check whether any of these companies' names or their domain names occurred in applications scanned by the project. The ICSI Haystack Panopticon project gathers data communication between mobile applications and trackers, and offers an online interface[5] to search for tracking

[5] `https://haystack.mobi/panopticon/index.html`, providing data from 2015.

activity on the collected dataset. We found eleven applications communicating with ThreatMetrix, but none that communicated with either BlueCava or Iovation. We downloaded these eleven applications and disassembled them in order to verify that these applications actually included a library or code that would communicate with ThreatMetrix. Via a thorough manual examination of the source code, we identified the presence of a ThreatMetrix library with the package name "`com.threatmetrix.TrustDefenderMobile`" in each application. Moreover, after gradually de-obfuscating each piece of code, we were able to confirm that the library is indeed performing fingerprinting by gathering a large amount of information about the device and transmitting it to their back-end. From this, we learned how the fingerprinting library communicates to its back-end servers.

To find fingerprinting libraries by Iovation and BlueCava, we downloaded all top free apps for each of the 62 categories on the Google Play Store (about 600 per category). This resulted in a dataset containing a total of 30,696 apps, collected in mid-July 2017. We scanned their source code for the string *"iovation"* and *"bluecava"*, respectively, which uncovered ten applications including Iovation. Upon manual investigation and de-obfuscation, we found they were indeed using a fingerprinting library by Iovation with the package name "`com.iovation.mobile.android`". We also found that Iovation's fingerprinting library avoids detection by the ICSI Haystack project because communication with Iovation's back-end is handled by the developer's back-end and not by the library itself (cf. Sec. 5.3).

Through reverse engineering, we found that unlike desktop browser fingerprinting, the identified libraries do not engage in side channel techniques to establish an identifier. That is, we did not find canvas fingerprinting or other similar techniques that are common to browser fingerprinters. Moreover, to our surprise we discovered that, except for the JavaScrip based plugin enumeration performed by ThreatMetrix, the libraries did not reuse code from their already existing browser fingerprinting libraries. Instead, the libraries seem to focus on the Android environment and to have been created independently to collect a large set of different attributes.

Note that there are several attributes on mobile phones that provide a globally unique identifier. Any of these would suffice as a way to identify a user without using fingerprinting. Hence, an app or library that goes beyond these and engages in promiscuous collection of diverse attribute data is thus suspect of performing fingerprinting.

4 FP-Sherlock: Hunting for Unknown Fingerprinters

To identify unknown fingerprinters, a definition of what constitutes a fingerprinter is needed. However, there is no agreed-upon definition of fingerprinting. Fingerprint-detecting tools such as FPDetective focus on detecting use of specific techniques, e.g. font list probing or canvas fingerprinting. In contrast, we aim to create a tool that can detect siphoning of large quantities of attributes that together constitute a fingerprint (as shown by Eckersley [3]). However, in the mobile domain there is no equivalent to the work of Eckersley. Previous studies focused on a subset of attributes (e.g. [15]), which only have a very limited overlap with attributes used by actual fingerprinters. This means we cannot determine whether a given set of attributes constitutes a unique fingerprint in the mobile domain. Thus we have to approach detection of fingerprinters differently. Instead of looking for a library that is fingerprinting, we look for fingerprinting behaviour: is the app collecting attributes commonly used by fingerprinters, and is this accumulation due to one single library?

Based on previous studies in browser fingerprinting [3,9,1,13], studies in mobile fingerprinting [12,4,5,15] as well as our findings from analysing mobile fingerprinters (cf. Section 3), we find that all fingerprinters share two characteristics:

Diversity: measures how many different attributes are accessed by a library.
Accumulation: determines whether the attributes identified by the diversity measure are all collected at one point in the library.

Both diversity and accumulation are necessary characteristics of fingerprinters, but they are not sufficient to positively identify a fingerprinter. Therefore, any approach based on them requires post-processing, e.g. manual inspection of found candidate fingerprinters.

To concretise the notion of diversity, we set out to construct a taxonomy of attributes. Diversity could then be based not on the amount of attributes itself, but on the amount of groups in the taxonomy accessed by the suspect library. We chose an initial set of classes of attributes based on the classifications proposed by Wu et al. [15], expanding it to incorporate the results of our analysis on the commercial fingerprinting libraries by Iovation and ThreatMetrix (cf. Table 5). To concretise the notion of accumulation, we examine the call graph (a directed graph) of a library. This allows us to perform flow analysis, determining whether there is a single point in the suspect library where attribute values could be accumulated. We thus arrive at the following concretisation:

δ-diversity: there must be at least one node in the call graph of the library that accesses attributes (either directly or accumulated) from at least δ different categories listed in Table 5,
α-accumulation: a node inherits the attributes accessed by its children, iff:

1. there exists a calling relationship between the two nodes (e.g. the parent node calls a subroutine of the child node)
2. the child node accesses at least one of the attributes listed in Table 5
3. the similarity between the class names of the parent and the child is at least α

4.1 System Design

FP-Sherlock is a static analysis tool for Android applications that uncovers fingerprinting libraries. As FP-Sherlock is based on static analysis, it is quick in comparison to approaches based on dynamic analysis. This enables large scale

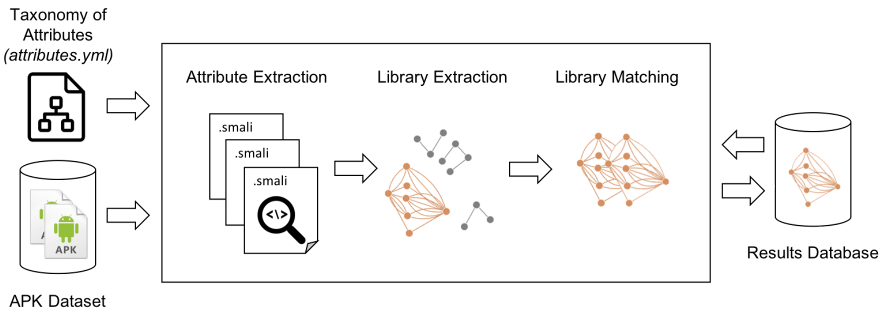

Fig. 1: FP-Sherlock's main steps and workflow.

scanning of applications. The tool is based on our characterisation of fingerprinting: *accumulation* and *diversity*. As mentioned, this characterisation is sound but not complete. FP-Sherlock is intended to be used to scan a large set and identify a much smaller set of candidates, which is then investigated by other, more time-consuming analysis methods (e.g. manual analysis or dynamic analysis). By updating the list of attributes covered and setting rates for α-accumulation and δ-diversity, FP-Sherlock can easily be adopted to other studies. FP-Sherlock's workflow in evaluating a single application is as follows (cf. Figure 1):

1. **Attribute Extraction:** the application is disassembled and the source code is examined for methods and fields that occur in the attribute taxonomy. In addition, FP-Sherlock keeps track of all call relationships between classes.
2. **Library Extraction:**
   (a) a partial call graph is generated, taking only call relations into account between classes that share a similar class name and that call on attributes.
   (accumulation)
   (b) we extract all connected components from the call graph. Each connected component represents a library and is only considered relevant if the overall categories of its attributes are sufficiently distinctive.
   (diversity)
3. **Library Matching:** every extracted library is matched with previously stored libraries, possibly exposing the un-obfuscated class names of obfuscated libraries. Eventually, the library is stored in a database for subsequent matches.

The remainder of this section explains each of the three main steps of FP-Sherlock in more detail.

Attribute Extraction. As input, FP-Sherlock takes a set of Android applications to be analysed and a taxonomy of attributes to look for in these applications. We created a taxonomy of 120 attributes, mapping all the smali methods and fields that we came across during our reverse engineering of Iovation's and ThreatMetrix's libraries (see Section 5) to our own representation of attributes.

We stored our taxonomy using YAML, a human-readable data serialization language commonly used for configuration files. In the attribute representation, we store class name, method name, field name and parameters separately. For example, the following smali method invocation:

```
const−string v1, ”http_proxy”

invoke−virtual {v0, v1}, Landroid/provider/Settings$Secure;−>getString
(Ljava/lang/String;)Ljava/lang/String;
```

is mapped to the subsequent YAML description:

```
- className: Landroid/provider/Settings$Secure
  methodName: getString
  parameters: [http_proxy]
  category: "Proxies"
  description: "HTTP Proxy Settings (Secure)"
```

We separate these in order to easily search for occurrences and overcome simple Java reflection, where methods of a class are retrieved by passing the name of the method as a string.

Every attribute contains a description and a category. We grouped the 120 attributes into 10 different categories (see Table 5 for more details). Our classification is based on the classification of 38 attributes by Wu et al. [15] (*Hardware*, *Operating System*, and *Settings*), which we expanded further to account for the attributes collected by ThreatMetrix and Iovation. We grouped attributes with similar purpose into one class, and separate classes based on perceived use. For example, unlike Wu et al., we distinguish a class *Localisation* as separate from *Settings*. This ensures that applications that access localisation information but no other settings score lower than apps that gather both localisation attributes and other settings. Similarly, we distinguish the class *Device* from *Hardware* and *Network* from *Carrier*. In addition, we identify a class *Applications* whose attributes pertain to infer installed applications. Finally, we consider the class *WebView*, which governs attributes accessed via JavaScript inside a Webview.

The analysis of an application starts by disassembling its bytecode using *baksmali*, a tool that translates DEX bytecode into a more human readable representation called *smali*. As a next step, we iterate over all method definitions. To boost performance, we slightly modified *baksmali* in order to keep the smali representation of a method definition in memory as a string, instead of writing it to a file. Afterwards, we use a regular expression (`const-string.*? v.+?, (".*?")`) to extract all the string constants contained inside the method definition. In addition, we keep track of calling relations between methods and classes, (deliberately ignoring system libraries, e.g. classes that start with `Landroid/`, `Lcom/android/` or `Lcom/google/`), by using another regular expression (`"invoke .+? {.+?}, (.+?;)"`), that extracts all method invocations. Finally, for every attribute, we search for occurrences of its class name and method name (or field name) within the method definition. Analogous, for the parameters of a method, we loop through the string constants and search for strings that match the given parameters. If the current class contains any occurrences of attributes, we create new node in the call graph labeled by the name of that class, and add all the matched attributes to it. Please note that the call graph is only a partial call graph as it only contains class nodes that hold attributes.

Library Extraction. In the previous step we extracted the attributes and added nodes to the call graph. A node represents a class and contains a list of attributes that have been found in the method definitions of the class. However, the previous step does not connect any of these nodes. In this step, we iterate over the pairs of the previously extracted call relations by solely considering pairs where the callee node contains at least one attribute and adding a connection to the call graph if the class names of the nodes are different, yet share at least a similarity score of α. The similarity between two class names is computed using fuzzy string matching. We use the JavaWuzzy[6] implementation. JavaWuzzy uses the Levenshtein Distance to determine the similarity between two strings. This metric determines the similarity of two strings by looking at the minimum number of edits required for two strings to be equivalent. However, this approach does not work for pair of nodes with short obfuscated class names, as these will not fulfil the similarity score of at least α. Hence, for such pairs we perform a less precise matching, where we check whether the token set ratio and the token set partial ratio of their class names are equivalent, and whether both class names share a non-empty longest common starting substring. We then add a connection between a pair of nodes, if these two properties are fulfilled. JavaWuzzy computes the token set ratio by tokenising the strings and comparing their intersection and remainder. The partial token set ratio is identical to the token set ration, except that the partial token set ratio matches based on the best substrings (e.g. the best partial match from a set of matching tokens), whereas the token set ratio matches based on pure Levenshtein Distance.

Finally, after adding the connections to the call graph, we extract the connected components. The connected components of a graph are the set of largest subgraphs of which every subgraph is connected, where connected means that there is is a path from any point in the graph to any other point in the graph. As a result, a connected component represents a library, e.g. a subgraph that is connected and yet independent from the other subgraphs, similar to the concept of a library included inside an Android application. For every connected component we compute the cardinality-based cosine set similarity between the set of all possible categories and the set of categories for the node of the connected component with the highest number of attributes (including its children's attributes). The cardinality-based cosine set similarity is computed as follows, where A and B are sets of categories:

$$cos(A,B) = \frac{|A \cap B|}{\sqrt{|A| \times |B|}}$$

A connected component must have a cosine set similarity score of at least δ in order to be considered as a library with potential fingerprinting-alike behaviour.

[6] https://github.com/xdrop/fuzzywuzzy

Limitations. Obviously, as FP-Sherlock is a static analysis tool that relies on comparing strings, it inherits certain limitations such as dynamic code loading or string encryption. Although, none of the studied fingerprinting libraries made use of such techniques at the time of writting.

Library Matching. In this last step, we compare the extracted libraries with previously analysed libraries in order to search for identical libraries. This can be useful in order to match obfuscated with non-obfuscated libraries and extract their non-obfuscated class names. We use a greedy library matching algorithm (see Algorithm 1) in order to compare two libraries. As previously stated, libraries are represented as connected components, hence graphs. The algorithm matches libraries by matching nodes. We match nodes based on attributes, the attributes of both nodes' parents and the attributes of both nodes children. We did not consider graph isomorphism as we want to preserve permutation between nodes. If a graph matches a previously analysed graph, we extract the longest common starting substring amongst the class names of its nodes and append it to the list of candidate names for that particular graph. If there is no match, we store the graph and create a new list of candidates names for that particular graph, appending its longest common starting substring as a first candidate. We store the graphs and the lists of candidate names in a MongoDB database for subsequent analysis.

4.2 Experiment: Scanning 30,000 Applications

As mentioned above, we downloaded a set of free top-rated applications from the Google Play store (∼30,000 apps) as of July 2017. We scanned this corpus with FP-Sherlock, using $\delta = 0.75$ and $\alpha = 0.75$. We ran a single instance of FP-Sherlock on a customer grade computer with a dual core CPU and 16 GB of RAM. With these settings we were able to scan our corpus in approximately 8 hours (∼1 second per app). We uncovered 150 candidate fingerprinting libraries. Of these, we analysed the six most popular ones in terms of occurrences in our corpus: *Amazon Mobile Ads*, *Chartboost*, *INFOnline*, *Kochava*, *Kontagent* and *Tapjoy*. More details about these six libraries, are condensed in Table 1.

Table 1: List of most popular libraries and their package names.

Library	Category	Package Name
Amazon (AZ)	Ads/Analytics	`com.amazon.device.ads`
Chartboost (CB)	Ads/Analytics	`com.chartboost.sdk`
INFOnline (IN)	Ads/Analytics	`de.infonline.lib`
Kochava (KC)	Ads/Analytics	`com.kochava.android`
Kontagent (KA)	Ads/Analytics	`com.kontagent.fingerprint`
Tapjoy (TJ)	Ads/Analytics	`com.tapjoy.connect`

Algorithm 1 A greedy library matching algorithm

```
function matchGraphs($G_1, G_2$)
    if $|G_1.nodes()| \neq |G_2.nodes()|$ or $|G_1.edges()| \neq |G_2.edges()|$ then
        return False
    matched $\leftarrow \{\}$
    for every node $n_1 \in G_1.nodes()$ do
        found $\leftarrow$ False
        for every node $n_2 \in G_2.nodes()$ do
            if $n_2 \notin$ visited and $n_1.attributes() = n_2.attributes()$ then
                if matchNodeList($n_1.parents(), n_2.parents()$) and
                  matchNodeList($n_1.children(), n_2.children()$) then
                    matched $\leftarrow$ matched $\cup \{n_2\}$
                    found $\leftarrow$ True
                    break
        if not found then
            return False
    return True

function matchNodeList($N_1, N_2$)
    if $|N_1| \neq |N_2|$ then
        return False
    matched $\leftarrow \{\}$
    for every node $n_1 \in N_1$ do
        found $\leftarrow$ False
        for every node $n_2 \in N_2$ do
            if $n_2 \notin$ matched and $n_1.attributes() = n_2.attributes()$ then
                matched $\leftarrow$ matched $\cup \{n_2\}$
                found $\leftarrow$ True
                break
        if not found then
            return False
    return True
```

5 Analysis of Fingerprinting-alike Libraries

We manually analysed the source code of every library using JADX[7], a DEX to Java decompiler with a GUI. For each of the analysed libraries, both those by fingerprinters as well as those uncovered by FP-Sherlock, we determined which permissions are necessary (cf. Table 2). Note that permissions with protection level "normal" are automatically granted by the system, whereas those with protection level "dangerous" must be actively granted by the user.

Once a library is called upon to gather a fingerprint, it typically operates in two or three phases, as shown below. The remainder of this section is structured along these three phases.

[7] https://github.com/skylot/jadx

Table 2: Permissions required by the analysed libraries.

Permission	Protection	IO	TM	TJ	AZ	IN	CB	KA	KC
`INTERNET`	*Normal*	✓	✓	✓	✓	✓	✓	✓	✓
`READ_PHONE_STATE`	*Dangerous*	✓	✓	✓	✓	✓	✓	✓	✓
`ACCESS_WIFI_STATE`	*Normal*	✓		✓	✓	✓	✓	✓	
`ACCESS_NETWORK_STATE`	*Normal*			✓	✓	✓	✓		
`ACCESS_COARSE_LOCATION`	*Dangerous*	✓	✓		✓				✓
`ACCESS_FINE_LOCATION`	*Dangerous*	✓	✓		✓				✓
`BLUETOOTH`	*Normal*	✓							
`GET_ACCOUNTS`	*Dangerous*	✓							

1. An optional initialisation phase, during which additional configuration may be loaded from a remote server;
2. A collection phase during which attribute values are collected; and
3. A submission phase, during which the accumulated fingerprint is transmitted back to a designated collection point, typically a server of the party that developed the library.

5.1 Initialization Phase

The ThreatMetrix library is the sole library (of those investigated) that contacts an external server before gathering attributes. It requests an XML file containing configuration information. This seems to be a way to include experimental attributes, such as novel checks for packages which indicate if the device has been rooted or to test for emulators by giving a list of phone numbers and IMEI numbers (a per-device unique number) known to be used in emulators.

5.2 Collection Phase

This section discusses our findings on the collection of attributes by the eight studied libraries and compares them to Wu et al.'s permissionless fingerprinter [15], which was based on implicit identifiers. The attributes collected by the libraries are shown in Table 5.

Inclusion of other fingerprinting libraries. During our manual analysis of Tapjoy's source code, we noticed that Tapjoy includes the ThreatMetrix fingerprinting library and combines it together with its own fingerprint.

Collection of explicit identifiers. Wu et al. [15] focused on implicit identifiers and therefore did not consider explicit identifiers. However, as one can see in Table 5, known fingerprinting companies such as Iovation and ThreatMetrix make extensive use of identifiers such as the Android ID, IMEI, Bluetooth and WiFi MAC addresses, etc. Note that not all explicit identifiers necessarily require a permission. For example, ThreatMetrix, Tapjoy and Amazon obtain the device hardware serial via `android.os.Build.SERIAL`, whereas access to this identifier does not require any permission while it can be used as a unique device identifier.

Attributes relevant for cross-device tracking. Of particular interest is Iovation's access to certain attributes: WiFi's SSID (network name), BSSID (MAC address of the base station), list of user accounts, phone number, subscriber ID and SIM serial number. Especially user account information may contain personally identifiable information (PII), such as username, that may link the device to external accounts (such as Facebook, Google, etc.). Obviously, all this information together can be used for cross-device tracking.

Context checks. Iovation and ThreatMetrix have some checks that consider the integrity and the environment of its hosting application. For instance, ThreatMetrix gathers some attributes in two different ways (via Java API and JNI) enabling them to detect inconsistencies. Both libraries check if the device is rooted or running inside an emulator by checking for instance the existence of installed binaries such as `/system/xbin/su`. Interestingly enough, Iovation search for installed applications that have the ACCESS_MOCK_LOCATION permission. This permission allows an application to override the current location with a fake location. Moreover, Iovation checks the signature on the hosting application. If this was re-signed, this probably indicates the hosting application has been potentially modified. Also, Iovation checks the `isUserAMonkey()` method. This method returns *true* if the user interface is undergoing structured testing by a program (e.g. a 'monkey'). Finally, Iovation checks wether the hosting application has the debug flag set, which should not occur in production runs.

Location fallback methods. In Table 5 we see that Iovation requests the cell ID and the location area code (*Localisation*: GSM/CDMA CID & LAC). Combined with the locale country and operator name, these attributes allow Iovation to derive a so-called "Global Cell-ID". This ID determines the specific cell in which a mobile device is located world-wide, without relying on GPS.

Device uptime. Iovation and ThreatMetrix derive the timestamp when the device was last booted, by subtracting the device's uptime from the current time. We suspect that this is because these companies found the timestamp of the last boot to have a high entropy. Note that we cannot verify this suspicion, since to the best of our knowledge, none of the published works on attribute entropy in Android devices consider device uptime.

Browser properties. ThreatMetrix gathers a list of installed browser plugins and mime-types via Javascript injected inside a WebView. This is rather bizarre as we suspect WebViews to have no browser plugins installed and to share the same set of mime-types.

Inference of installed applications. Iovation abuses a method intended as an easter egg by Google (i.e. `UserManager.isUserAGoat()`) in order to check if a specific goat simulator package is installed. ThreatMetrix scans every 60 seconds for newly installed non-system applications on the device as well as a list of running processes. Tapjoy checks whether two alternative application markets are installed on the device: Gfan (a Chinese application market) and SK Telekom (a South Korean application market). Tapjoy also checks for the presence of four specific social sharing applications: Facebook, Twitter, Google+ and LinkedIn, thus inferring in which social networks the user is present.

Proxy detection. Iovation tries to detect if the user is using a proxy for FTP, HTTP, and HTTPS. It does so by calling the method `ProxySelector.getDefault().select(<url>).toString()`, for the three URLs: *'ftp://www.example.com/'*, *'http://www.example.com/'* and *'https://www.example.com/'*.

Font enumeration. ThreatMetrix attempts to get the list of system fonts, first via JNI, failing that, via Java, looking in the `/system/fonts` directory for all files having the `.ttf` extension. Note that this is not equivalent to font list probing – in desktop browsers, the font list is not directly accessible, but may be partially determined by checking whether specific fonts are present. Conversely, on Android systems, the list of fonts is directly accessible. Moreover, Wu et al. found that the lists of fonts in Android devices does not provide a great amount of entropy, in contrast to font lists on desktop computers.

Storage capacity. Similar to Wu et al., Iovation and ThreatMetrix compute the capacity of the internal and external storage by multiplying block size with block count. ThreatMetrix goes further by also computing the available space left on the internal storage.

Battery characteristics. Iovation gathers information about the current battery level and whether the device is currently plugged to a power source. Olejnik et al. [10] have shown that the HTML5 Battery Status API exposes a fingerprintable surface that can be used to track web users in short time intervals.

CPU and Memory characteristics. Iovation and ThreatMetrix, both gather information about the CPU and Memory by reading `/proc/cpuinfo` and `/proc/meminfo` respectively. Moreover, Threatmetrix extracts the BogoMips value per CPU core. The BogoMips (from 'bogus' and MIPS) is an unscientific measurement of CPU speed made during boot.

Camera characteristics. Iovation gathers characteristics of all built-in cameras. In particular, it gathers the values of INFO_SUPPORTED_ HARDWARE_LEVEL and SENSOR_CALIBRATION_TRANSFORM1 for every camera.

5.3 Submission Phase

After collection, the gathered data is sent back to the back-end. Thankfully, all studied libraries make use of HTTPS to submit the gathered data. Nikiforakis et al. [9] found two scenarios for communicating browser fingerprints: by an included third party without involvement of the first party, or explicitly upon the request of the first party. In our analysis, we find exactly the same two scenarios in the mobile domain. While the top six investigated libraries do not require interaction with the host application developer, the libraries by ThreatMetrix and Iovation take a different approach. These two libraries operate on the explicit request of the hosting application, and provide the gathered user identity to the hosting application. In contrast, the top six libraries are advertising and analytics libraries that do not offer explicit identification services to applications, and therefore do not have to communicate an identity back to the developer. In the remainder of the submission section, we thus focus on the submission processes of Iovation and ThreatMetrix.

Iovation. Iovation is the odd library out, as it does not explicitly submit the fingerprint to its own back-end, but leaves this to the app developers. These thus have to set up and maintain a back-end server which gathers and forwards the fingerprints to Iovation's back-end. This also explains why Iovation was not in the ICSI Panopticon dataset. Iovation's fingerprinting library encrypts its fingerprints (using AES in CBC mode without padding) via a hard-coded key that is identical across all applications that include Iovation. Moreover, Iovation uses a random initialisation vector, which is concatenated in clear to the AES output. The input to AES is a concatenation of all the collected attributes, starting with the magic number "0500". In order to collect a fingerprint, the hosting application typically calls Iovation's method `DevicePrint.getBlackbox()`. This method returns a base64 encoded version of the encrypted string. Application developers explicitly do not know about the AES encryption, nor about the key used for encryption. Thus, app developers must submit the collected fingerprints back to Iovation if they are to be used. This delivery mechanism not only permits Iovation to hide their implementation details and to bill their customers, but this also allows Iovation to remain undetected towards traffic analysis tools such as the ICSI Panopticon.

ThreatMetrix. In contrast to Iovation's fingerprinting library, ThreatMetrix communicates its fingerprint directly to its back-end servers. The fingerprint is submitted via a web beacon, a 1x1 pixel image hosted at *https://h.online-metrix.net/fp/clear.png.* ThreatMetrix follows the same approach to submit fingerprints via its browser fingerprinting scripts. The body of the request contains the fingerprint, as the parameter "ja". The fingerprint is "encrypted" by XOR'ing it with the session_id. The HTTP Referer header contains the package name of the host application. In addition to submitting the fingerprint, ThreatMetrix also submits an HTTP cookie (thx_guid) to its own back-end servers. This cookie includes a unique identifier that is based on the Android ID of the mobile device. The developer can request any information about the user from ThreatMetrix's back-end servers via the session_id.

6 Adoption of Investigated Libraries

We investigated the adoption of the investigated libraries across popular Android applications. Fingerprinting-alike behaviour turns out to be much more common

Table 3: Prevalence of fingerprinting-alike libraries in popular applications from the Google Play Store.

		# Installs			
Library	**# Apps**	**[0, 10K)**	**[10K, 100K)**	**[100K, 1M)**	**[1M, $+\infty$)**
Chartboost	4493	188	633	1564	2108
Amazon	1428	85	204	492	647
Tapjoy	1204	42	132	350	680
ThreatMetrix	471	3	22	123	323
Kochava	221	4	12	49	156
INFOnline	220	18	66	78	58
Kontagent	54	0	0	15	39
Iovation	12	0	1	1	10
	5917 *unique*				

in Android applications than on the web (see e.g. Acar et al.'s study [1]) We found 5.917 unique applications that include at least one of the eight studied libraries, hence 19% of our dataset. Our findings are summarised in Table 3. Using the classification from the Google Play Store, we state that the majority of the investigated fingerprinting libraries can be found inside games with the following categories: *Action*, *Casino*, *Casual*, *Racing* and *Games*, whereas the five app categories with the least library occurrences are: *Events*, *Art & Design*, *Business*, *Maps and Navigation* and *Libraries & Demo*.

Table 4: Prevalence of identified libraries in applications subject to COPPA.

Library	**# Apps Subject to COPPA**
Chartboost	13
Tapjoy	6
Kochava	5
Amazon Mobile Ads	3
ThreatMetrix	2
INFOnline	0
Iovation	0
Kontagent	0
	21 *unique*

Adoption Amongst Apps Targeted at Children. Collecting children's personal information without parental consent is illegal in the USA under the *Children's Online Privacy Protection Act* (COPPA) [2]. In particular, collection of personally identifiable information of children is mostly prohibited, and exceptions are only allowed under "verifiable parental consent". Where a recent study by Reyes et al. [11] searched for any COPPA violations in their data set, we explicitly limit ourselves to the previously analysed libraries. Therefore, we do not need to encode COPPA regulations into a scanner.

We identified 21 applications from Play Store categories *5 & Under* and *6–8* (both clearly subject to COPPA) that used one or more of the analysed libraries. Only three of the eight investigated libraries were not present (cf. Table 4).

The question of whether or not the observed behaviour is a violation of COPPA is a legal matter beyond the scope of this paper. That notwithstanding, we hold the view that apps explicitly aimed at children should sidestep this question by not engaging in fingerprinting-alike behaviour at all.

7 Conclusions and Future Work

We reverse-engineered two commercial fingerprinting libraries for Android apps and analysed their behaviour. We expected these libraries to reuse techniques that their creators apply for fingerprinting browsers but, to our surprise, we did not encounter this. We also expected to find typical fingerprinting techniques such as canvas fingerprinting, but found no such techniques. These commercial authentication libraries apparently do not need such techniques to be certain about a user's identity – collecting a large set of attribute values and some unique identifiers apparently suffices. This implies that there is quite some authenticating information to be gleaned from the collected attribute values.

With this in mind, we set out to identify other libraries that gather similar amounts of attribute values. We designed and implemented *FP-Sherlock*, a static scanner for fingerprint-alike behaviour. FP-Sherlock is based on the notions of *diversity* and *accumulation*, which provide a necessary but incomplete characterisation of fingerprint behaviour.

We applied FP-Sherlock to a corpus of ∼30,000 applications and found several candidate fingerprinters. Of these, we reverse-engineered the six most popular libraries (i.e., with the highest usage rate in our corpus). We were able to establish a lower bound on mobile device fingerprinting-alike behaviour that is vastly higher than browser fingerprinting: 5,917 out of 30,695 or 19.28%. In contrast, recent studies into browser fingerprinting prevalence find between 0.4% and 1.4% adoption rate amongst popular websites. While the found behaviour lacks the telltale signs of fingerprinting present in browser fingerprinters, the amount of data gathered by these libraries is clearly unwarranted and exceeds the bounds of reason. We believe that neither users nor app developers are aware of this data gathering, let alone of the scale of this.

Moreover, we investigated how many apps targeted at children include one of the studied libraries. Tracking children is (in general) subject to stricter legal restrictions than tracking adults. In our dataset, we found 21 apps targeted at children that included one or more of the studied libraries. Two of the used libraries explicitly fingerprint, and thus definitely should fall under tracking restrictions.

Future work. Future work focuses on three aspects. First of all, we are currently developing a framework to repeat Eckersley's study of attribute entropy [3] for mobile devices. Secondly, we are looking to improving and automating the detection of fingerprinters, using machine learning techniques. Thirdly, countering fingerprinters seems more challenging than for web browsers, as some fingerprinters piggyback on the first-party. Solution approaches based on guided randomization at level of the Android API should be further investigated.

References

1. Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., and Preneel, B. FPDetective: dusting the web for fingerprinters. In *Proc. 2013 ACM SIGSAC conference on Computer & communications security (CCS'13)* (2013), ACM, pp. 1129–1140.
2. Commission, F. T., Commission, F. T., et al. Children's online privacy protection rule (COPPA).
3. Eckersley, P. How unique is your web browser? In *Proc. 2010 Privacy Enhancing Technologies conference (PET'10)* (2010), vol. 6205, Springer, pp. 1–18.
4. Hupperich, T., Maiorca, D., Kührer, M., Holz, T., and Giacinto, G. On the robustness of mobile device fingerprinting: Can mobile users escape modern web-tracking mechanisms? In *Proc. 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015), ACM, pp. 191–200.
5. Kurtz, A., Gascon, H., Becker, T., Rieck, K., and Freiling, F. Fingerprinting mobile devices using personalized configurations. *Proceedings on Privacy Enhancing Technologies (PETS'16) 2016*, 1 (2016), 4–19.
6. Laperdrix, P., Rudametkin, W., and Baudry, B. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *Proc. 2016 IEEE Symposium on Security and Privacy (S&P'16)* (2016), IEEE, pp. 878–894.
7. Mayer, J. R., and Mitchell, J. C. Third-party web tracking: Policy and technology. In *Proc. 2012 IEEE Symposium on Security and Privacy (S&P'12)* (2012), IEEE, pp. 413–427.
8. Mowery, K., and Shacham, H. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of 2012 workshop on Web 2.0 Security and Privacy (W2SP'12)* (2012), IEEE, pp. 1–12.
9. Nikiforakis, N., Kapravelos, A., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proc. 2013 IEEE Symposium on Security and privacy (S&P'13)* (2013), IEEE, pp. 541–555.
10. Olejnik, Ł., Acar, G., Castelluccia, C., and Diaz, C. The leaking battery. In *Proc. 2015 Workshop on Data Privacy Management (DPM'15)* (2015), Springer, pp. 254–263.
11. Reyes, I., Wijesekera, P., Reardon, J., On, A. E. B., Razaghpanah, A., Vallina-Rodriguez, N., , and Egelman, S. "won't somebody think of the children?" examining coppa compliance at scale. *PoPETs 2018*, 3 (2018), 63–83.
12. Spooren, J., Preuveneers, D., and Joosen, W. Mobile device fingerprinting considered harmful for risk-based authentication. In *Proc. Eighth European Workshop on System Security (EuroSec'15)* (2015), ACM, pp. 6:1–6:6.
13. Torres, C. F., Jonker, H., and Mauw, S. FP-Block: usable web privacy by controlling browser fingerprinting. In *Proc. 2015 European Symposium on Research in Computer Security (ESORICS'15)* (2015), Springer, pp. 3–19.
14. Vallina-Rodriguez, N., Sundaresan, S., Razaghpanah, A., Nithyanand, R., Allman, M., Kreibich, C., and Gill, P. Tracking the trackers: Towards understanding the mobile advertising and tracking ecosystem. *CoRR abs/1609.07190* (2016).
15. Wu, W., Wu, J., Wang, Y., Ling, Z., and Yang, M. Efficient fingerprinting-based android device identification with zero-permission identifiers. *IEEE Access 4* (2016), 8073–8083.

A A Detailed Taxonomy of Attributes Used by Fingerprinting-alike Libraries

Table 5: A detailed taxonomy comparing all the attributes used by the studied libraries.

			Authentication		**Advertising/Analytics**					
Category	**Attribute**	**[15]**	**IO**	**TM**	**TJ**	**AZ**	**IN**	**CB**	**KA**	**KC**
Applications	List of Installed Applications	✓	✓	✓ (JAVA/JNI)	✓					✓
	App/Package Name		✓	✓	✓	✓	✓	✓	✓	✓
	App/Package Version		✓		✓	✓	✓	✓	✓	✓
	App/Package Hash			✓	✛					
	App/Package Signatures		✓							
	Debug Flag		✓							
	Is User a Goat/Monkey		✓							
	List of Special Files & Properties			✓ (JAVA/JNI)	✛					
	List of Running Processes			✓ (JAVA/JNI)	✛					
	Usage of Alternative App Markets				✓					
	List of Social Sharing Services				✓					
Carrier	SIM Operator Name (P)		✓				✓	✓		
	SIM Operator Country (P)		✓	✓	✛			✓		
	Network Operator Name (P)		✓	✓	✓	✓	✓	✓	✓	✓
	Network Operator Country (P)		✓		✓			✓		
	MSISDN (Phone Number) (P)		✔							
	IMSI (Subscriber ID) (P)		✔							
	SIM Serial Number (P)		✔							
Device	Manufacturer	✓	✓	✓	✓	✓	✓	✓		
	Brand		✓	✓	✛		✓			✓
	Model	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Product		✓	✓	✛		✓	✓		
	Device Build		✓	✓	✛		✓		✓	
	IMEI (Device ID) (P)		✔	✔	✔		✔			
	Uptime		✓	✓	✛					
Hardware	Screen Resolution	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Screen Orientation		✓		✓	✓	✓	✓		
	Internal Storage Capacity	✓	✓	✓	✛					
	External Storage Capacity	✓	✓							
	Available Internal Storage			✓	✛					
	Battery Information		✓							
	Proximity Sensor		✓							
	CPU Information		✓	✓	✛					
	RAM Information		✓	✓	✛					
	Hardware Build						✓			
	Hardware Serial			✔	✔	✔				
	Camera Information		✓							
Identifiers	**Android ID**		✔	✔	✔	✔	✔	✔	✔	✔
	Google Play Advertising ID (P)				✔	✔	✔	✔	✔	✔
	Facebook Attribution ID								✔	✔
Localisation	Locale Country			✓	✓	✓	✓	✓		
	Locale Currency		✓							
	Locale Language	✓	✓	✓	✓	✓	✓	✓	✓	
	Timezone	✓	✓	✓	✛			✓		
	Time & Date Format	✓								
	Geolocation (P)		✓	✓	✛	✓				✓
	GSM/CDMA CID & LAC (P)		✓							

			Authentication		Advertising/Analytics					
Category	**Attribute**	**[15]**	**IO**	**TM**	**TJ**	**AZ**	**IN**	**CB**	**KA**	**KC**
Network	Local IP Addresses		✓						✓	
	Local Hostname		✓							
	Connection Type (P)				✓	✓	✓	✓		
	WiFi MAC Address (P)		✔		✔	✔	✔	✔	✔	
	WiFi SSID (P)		✓							
	WiFi BSSID (P)		✓							✓
	Bluetooth MAC Address (P)		✔							
	List of Available Proxies		✓							
Operating System	OS Name		✓	✓	✓	✓	✓	✓	✓	✓
	OS Version		✓	✓	✓	✓	✓	✓		✓
	Root Status	✓	✓(JNI)	✓(JAVA/JNI)	✢			✓		
	Is an Emulator		✓		✓			✓		
	Kernel Information	✓	✓							
	API Level		✓						✓	
	Build ID		✓							
	Build Display		✓	✓						
	Build Fingerprint		✓							
	Build Host		✓							
	Build Time		✓							
	User-Agent	✓		✓(JAVA/JS)	✢	✓				✓
	System Storage Structure	✓								
	Root Directory Structure	✓								
	Input Methods	✓								
	System Font Size	✓								
	List of System Fonts	✓		✓(JAVA/JNI)	✢					
Settings	Automatic Time Sync	✓								
	Automatic Timezone Selection	✓								
	Time of Screen Locking	✓								
	Notify WiFi Availability	✓								
	Policy of WiFi Sleeping	✓								
	Lock Pattern Enabled/Visible	✓								
	Phone Unlocking Vibration	✓								
	Sound Effects Enabled	✓								
	Show Password in Text Editors	✓								
	Screen Brightness Mode	✓								
	Is Device Orientation Locked	✓								
	Current Wallpaper	✓								
	List of Default Ringtones	✓	✓							
	Install Non Market Apps		✓							
	Granted Permissions		✓		✓					✓
	List of User Accounts (P)		✓							
	HTTP Proxy Settings		✓							
	System Volume									✓
Web View	List of Plugins			✓(JS)	✢					
	List of Mime-Types			✓(JS)	✢					

✔ : Explicit identifier that by itself is sufficient to uniquely identify a device.
✢ : Attribute collaterally called via the ThreatMetrix SDK.
(P) : Attribute requires a permission in order to be retrieved.
(JS) : Value retrieved via JavaScript Interface (allowing calls to JavaScript objects).
(JNI) : Value retrieved via Java Native Interface (allowing calls to native methods written in C/C++).
(JAVA/JNI) : Value retrieved via Java and Java Native Interface.
(JAVA/JS) : Value retrieved via Java and JavaScript Interface.