AndroPRINT: Analysing the Fingerprintability of the Android API

Gerald Palfinger
Secure Information Technology Center - Austria (A-SIT)
Graz, Austria
gerald.palfinger@iaik.tugraz.at

Bernd Prünster
Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology
Graz, Austria
bernd.pruenster@iaik.tugraz.at

ABSTRACT

In recent Android versions, access to various (unique) identifiers has been restricted or completely removed for third-party applications. However, many information sources can still be combined to create a fingerprint, effectively substituting the need for these unique identifiers. Until now, finding these fingerprintable sources required manually sifting through the API documentation to identify each information source individually. This paper presents AndroPRINT, a framework that automatically recognizes fingerprintable information sources on Android devices. For this purpose it automatically invokes methods, queries fields, and retrieves data from content providers. We show that this framework allows automating the elaborate task of finding such fingerprintable information sources in different experiments. In these experiments, a variety of information sources could be identified, which provide a vast amount of unique features for fingerprinting. Furthermore, AndroPRINT detected undocumented unique device identification features, which are a result of manufacturer adaptations. These vendor customisations even revealed personal data, such as the user's email address and cryptographic keys used for cross-device communication. The fact that this information can be retrieved without the user noticing means that vendor customisations can effectively defeat the tight permission system of modern smartphone operating systems.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**;

KEYWORDS

fingerprinting, Android API, automatic analysis, unique identifier

ACM Reference Format:
Gerald Palfinger and Bernd Prünster. 2020. AndroPRINT: Analysing the Fingerprintability of the Android API. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3407023.3407055

1 INTRODUCTION

Many mobile applications rely on personalised advertisements as a main stream of income [6]. To tailor these advertisements, users have to be recognised across applications. Many applications use fingerprinting to reidentify users [16]. However, with Android 10 Google has removed or limited access to many fingerprintable information sources [3]. This includes access to camera details, information about saved WiFi networks, and access to unique identifiers such as the IMEI or device serial number—some of which could previously be retrieved without the users granting access to these identifiers.

This work introduces *AndroPRINT*, a framework to automatically discover novel ways to fingerprint individual Android devices (and thus users) through the use of public API functionality. The general technique of fingerprinting is already known and is actively being exploited to track users on the Web through browser fingerprinting [4]. In essence, properties such as screen resolution or installed fonts can be queried by websites and if a script (such as an advertisement banner or social network integration) is present among many websites, individual users can be tracked across website boundaries. AndroPRINT applies this approach to the Android platform and is able to automatically discover functionality that can be used to create device-specific fingerprints. Once identified, cross-app user tracking is easily possible for any vendor of popular libraries, such as large advertisement networks, which are incorporated in a vast amount of Android apps. This is possible, because the queried fingerprints are app-independent. Contrary to querying a user's address book, for example, the invocation of API methods identified by AndroPRINT requires no user consent and thus happens undetected in the background.

While evaluating the efficacy of our approach, we discovered fingerprintable features that are perfectly stable (i.e. rebooting a device has no effect) and unique, thus making it possible to identify individual users with high probability. To provide one such example, the full list of ringtones can be queried. Since this also covers any ringtones added by the user—including the precise timestamp of loading any custom ringtones onto the device—identifying anyone using custom ringtones can be accomplished with high confidence. Moreover, we uncovered a flaw on Samsung devices, which enables any app to quietly retrieve the e-mail address associated with the user's Samsung account used on the device, as well as unique user and hardware IDs. This in itself already has grave consequences on user privacy, given that users not only cannot consent, but also cannot prevent this.

Approach. To find fingerprintable methods and fields in the Android API we use the following strategy:

(1) A smartphone application traverses through the Android API to collect as much data as possible from fields, methods, and content providers (a mechanism to pass data between applications). The application creates objects to fetch fields

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

and invoke instance-methods as needed. Constant values are pre-parsed from the API documentation in order to correctly invoke methods expecting parameters from a set of predefined constants.
(2) The application is executed on multiple smartphones to collect potentially fingerprintable data. This collection phase is carried out twice on each device to detect and remove data which is changing between application restarts and reinstallation.
(3) Utilising our analysis framework we then search the swath of collected data for fingerprintable information. In this regard, values which differ on the same device are cleaned from the data. The collected values which differ between devices are marked as fingerprintable. While not strictly needed for fingerprinting, we manually categorise the fingerprintable data to ease human interpretation of the results.

While smartphone fingerprinting has received some attention in the past [19], we are the first to present a fully-automated approach that does not require any a-priori knowledge about the semantics of the Android API. More details on our contribution are provided below.

Contribution. While the primary goal of AndroPRINT is automated identification of functionality which enables device fingerprinting, our contribution as a whole is fourfold:

(1) We present the first framework to automatically detect fingerprintable information on Android. We extend an existing tool to traverse the Android API. Using this approach we collect fields and return values which contain possibly fingerprintable information.
(2) We dump all accessible content providers referenced in the Android API, thus uncovering (possibly undocumented) ways of cross-app information sharing, which can also be exploited to enhance fingerprinting accuracy.
(3) We introduce an analysis framework which removes spurious data and analyses the collected data from multiple devices for fingerprintable information. We analyse the results and categorise the identified methods, fields, and content providers.
(4) We uncover previously unknown unique user and device identifiers which are a result of vendor customisations.

Outline. The rest of this paper is organized as follows: The following section discusses related work. Section 3 then provides insight into how we systematically gather information from the Android API and how we analyse this information. Afterwards, Section 4 discusses the coverage our approach achieves and lists the results. Based on this, Section 5 discusses our results, limitations of the approach, and potential countermeasures. Finally, Section 6 concludes this paper.

2 RELATED WORK

In the following paragraphs we discuss related work. Especially browser fingerprinting has sparked a lot of interest in the research community. Various papers have been written on the possibility of browser fingerprinting and mitigations against it. The following paragraph discusses the work on browser fingerprinting. In comparison, mobile device fingerprinting has received less attention. We introduce the existing work in the subsequent paragraph. Finally, we present related approaches which use automation to detect vulnerabilities on smartphones.

Browser Fingerprinting. Browser fingerprinting can be used to identify a user across different web pages without storing data, such as cookies, locally in the browser. Eckersley [4] has collected information, such as the user agent, HTTP headers, or browser settings to show that it is possible to fingerprint users. The paper shows that the collected fingerprints change over time. However, as only parts of the fingerprint change at one point in time the collected fingerprints can still be connected to a user if observed regularly. Eckersley shows that certain privacy enhancing browser extensions are actually detrimental to the users privacy as they improve fingerprintability. For example, an extension that changes the user agent to a unique string could allow to uniquely identify a user.

Cao et al. [1] show that systems can be fingerprinted using hardware and operating system peculiarities. They use differences in rendered pictures, varying feature support of the web audio API, and the number of logical cores to create a fingerprint. Such fingerprints allow user tracking across different browsers and even operating systems.

Starov and Nikiforakis [15] show that it is possible to fingerprint a browser based on the installed extensions. Most browsers do not allow direct access to the list of extensions. However, certain extensions alter the content of the DOM. Based on these changes the authors show that it is possible to infer the extension. Sánchez-Rola et al. [11] use a timing side channel to detect activated browser extensions. Depending on the time it takes to access the resources of an extension they can infer whether or not the probed extension is installed. Furthermore, leaked internal extension URLs allow them to detect further extensions.

Schwarz et al. [12] use an automated approach to detect differing JavaScript properties in browsers which enable inferring the underlying hardware. The authors argue that their approach allows them to automatically construct browser fingerprints and tailor microarchitectural attacks to the detected hardware.

In addition to detecting features which can be used to create browser fingerprints, work has also been done to hamper or thwart browser fingerprinting. The Tor browser reduces fingerprintability of their users by a combination of different techniques, such as employing letterboxing to only allow for coarse-grained scaling of the browser's window and disabling or spoofing certain local information sources that could be queried using JavaScript or CSS [10]. Firefox also adopted some of the anti-fingerprinting features spearheaded by the Tor Browser, which can be activated as part of the enhanced tracking protections [18].

FPGuard [5] tries to detect fingerprinting behaviour of websites using heuristics. It monitors access to fingerprintable information sources, such as the navigator object, canvas elements, or loaded fonts. If a website exhibits suspicious behaviour FPGuard tries to alter the fingerprint to avoid user identification. Similarly, PriVaricator [9] randomises fingerprintable information to avoid the linkability of fingerprints. FP-Block [17] randomises fingerprintable information, but does so only cross-domain to avoid breaking

certain functionality on websites. However, such changes can often be detected by searching for inconsistent or missing information [4, 12].

While the ways to identify users based on browser functionality are diverse, the general principle is always the same and it is also applicable to the environment of smartphone apps. Like JavaScript is restricted to using functionality implemented by browsers, Android apps consume APIs defined by Google (naturally, the same principle also applies to Apple's iOS) and possibly extended by the smartphone vendor (only applicable in the Android case). Thus the same approach used for browsers is also applied to smartphone fingerprinting as outlined below.

Smartphone Fingerprinting. Kurtz et al. [8] pick 29 configuration features from the iOS SDK to show that fingerprinting iOS devices using non-unique identifiers is feasible. The authors collect fingerprints from different devices and show that it is possible to re-identify returning users with high probability. Similarly, Wu et al. [19] manually identify 38 different information sources which can be used to fingerprint Android devices. The identified sources are evaluated using three different algorithms using a dataset of collected fingerprints. Torres and Jonker [16] search for fingerprinting behaviour in Android applications. They identified eight different advertising, analytics, and authentication libraries which exhibit fingerprinting behaviour. Related to fingerprinting, Jing et al. [7] collect various data from the file system, Android API, and system properties to differentiate emulated from real devices. They limit the collection of information from the Android API to a small subset of the APIs exposed by system services which do not require any parameters.

As AndroPRINT aims at automated detection of fingerprintable information, a different, more exhaustive approach is required. We therefore also briefly touch automated vulnerability detection, as our approach relies on similarly automated workflows.

Automated Vulnerability Detection on Smartphones. ProcHarvester [13] is a tool to automatically find side channels in the /proc file system of Android. It searches for changes in the statistics provided in the /proc file system to infer user actions. SCAnDroid [14] is a framework to detect side channels in the Android API. The framework automates the invocation of methods and identifies methods which react to different user actions. Both approaches use machine learning to assess the gathered data and infer the actions performed on the smartphone.

3 METHODOLOGY

AndroPRINT's core boils down to systematically issuing calls to the Android API from within an Android app that has been specifically created for this purpose. The information gathered this way is subsequently analysed off-device to actually interpret the gathered results with respect to fingerprintability of individual API calls. Automating API calls relies heavily on reflection. However, not all information required to issue some constructor or method calls (such as parameters that are only allowed to take some constant) can be extracted from the bytecode. In order to work around this limitation, the Android API documentation is parsed for such values which are then fed into the Android app.

This results in an overall three-tier process whose phases are called *preparation* (parsing API documentation), *collection* (automated invocation of API calls, and recording the results), and *evaluation* (actually interpreting the collected data). The fully automated nature of AndroPRINT's approach not only makes it possible to cover a vast amount of the Android API without manual intervention, but also results in a robust and sustainable approach that remains applicable even for future versions of Android.
The remainder of this section discusses these phases in detail and elaborates on how and why this produces tangible results.

3.1 Preparation Phase

As mentioned, some information that is not available on the device during runtime needs to be parsed from the publicly available Android API documentation. This is one key task of the preparation phase. In fact, however, we parse information about all documented classes, their methods including the parameter names, and all constructors with their parameters. The gathered data is fed into the Android app which is used to invoke API methods during the collection phase. The app uses this data to create class objects, invoke methods, and query fields. The basic parsing of Android API documentation and the smartphone application which collects the data is in parts based on the SCAnDroid framework [14]. It has been enhanced with support for field access, content providers, and improved coverage by handling constants and objects of derived types.

In addition to gathering the structure of the Android API, possible constants are parsed. These constants are usually defined in an HTML table specifying the parameters, their type, and a description of their usage. The usage description usually also contains possible constant values. However, it does not have a structured format. Thus, we define the text content of all hyperlinks and HTML code blocks as possible constants. As can be seen in the figure, this methodology may miss some constants which are located in the plain text. Therefore, we also search the remaining text for possible constants by including all words which contain dots. This broad selection of possible constants also includes statements which are either in fact not constants or for some other reason not interpretable. These invalid values, however, do not have an impact, as they are automatically discarded when fed into AndroPRINT's app during the collection phase.

3.2 Collection Phase

In the collection phase possible fingerprintable information from the Android API is gathered. In order to collect this data the return values of methods, field values, and content providers are aggregated. To invoke these methods and fetch the values of fields all of the classes of the Android API are systematically queried. The previously parsed list of classes and methods are used as a starting point for the collection. The classes in this list are queried via Java Reflection. All of the declared methods and fields in these classes will be invoked or fetched. This also includes public methods and any fields which are not described in the API documentation. Static methods and fields can be called or queried without creating a class object. For all non-static methods and fields a corresponding class object has to be created first. The following section shows how

these objects are created, how constant values are interpreted, and how the methods and fields have been selected.

3.2.1 Initialisation. In the initialisation process the parsed constant values are interpreted first. After that, the required class objects are created. Last, the methods are invoked to collect the return value and fields are queried.

Interpretation. To interpret the parsed constants and all predefined values, the *BeanShell*[1] script interpreter is used. BeanShell is fed the values as string representation and returns the interpreted values. The interpreted values are then stored for the upcoming phase. All non-interpretable constants are discarded in this step.

Object Creation. Invoking non-static methods or querying non-static fields requires the instantiation of an appropriate object. The object has to be of the type that implements the method. If a method takes parameters it is furthermore needed to create objects for these parameters. Parameter types can be elicited at runtime via the reflection API. If these are parameters of a primitive type such as `int` or `boolean`, the required objects can be created via the Java wrapper types such as `Integer` and `Boolean`. The values for these primitive types can be predefined individually for each parameter of each method. If no individual default value has been set, a global predefined value for each type, such as 0 for integers, will be used. To connect the predefined values with the invocation of the method the name of the parameter has to be known. However, the name of parameters cannot be queried at runtime. Thus, we utilise the data gathered during the preparation phase to connect parameters with their names. This allows us to predefine values such as `packageName` or `pid`, which are needed to successfully invoke certain methods.

To create objects of non-primitive type, their class's constructors will be fetched and invoked at runtime. The created objects will be saved for later use. If a constructor takes parameters, these will be created analogously to the method invocation process described before.

Method Selection and Invocation. To reduce side effects, only methods which are usually associated with data gathering are invoked. Therefore, the set of methods to be invoked is reduced to methods containing the following prefixes: *get*, *has*, *is*, *query*, and *support*.

These methods can either return values of primitive or non-primitive type. Return values of primitive type usually contain one piece of information. Non-primitive return values, in contrast, can comprise a plethora of information. They may also include further methods which can be invoked to extract information. Therefore, these two groups are treated separately. Any primitive values returned by a method are saved to a file for later analysis. However, if a method returns an object, the object will be explored. The process is equal to directly created objects, i.e., all methods and fields will be invoked or queried. Since this process can potentially continue endlessly, the framework stops when the maximum call depth is reached. For the experiments this call depth has been set to 3 as this represents a sweet-spot in the trade-off between accuracy and execution speed.

[1] https://github.com/beanshell/beanshell

The objects which have been returned from methods have to be affiliated to a parsed class first. This is required to associate predefined values and constants for parameter creation, in case of parameters that are expected to only take predefined values. In order to create this connection the name of the class is elicited via reflection. The framework then checks whether or not the class is existing in the list of parsed classes. However, this direct comparison only works if an object is of exactly the same type as a parsed class. In practice, however, these objects can also be of derived types which are not explicitly treated in the documentation. Therefore, we also check whether or not the object is derived from any of the parsed classes. If that is the case, we use the parsed information of the base class. Thus, all predefined parameters for methods of the base class can be used during invocation, while for additional methods of the derived class the default values for each data type is used. Otherwise, the default values are used for all methods. All of the gathered return values will be saved to a list for later analysis.

Field Evaluation. In addition to invoking methods, all of the accessible fields of the API will be queried. In order to query them, the objects created for the invocation of methods and the non-primitive return values of methods will be used. The value of these fields will also be saved in a list for later analysis.

Content Provider. A content provider on Android provides an abstract interface for accessing data. Similar to a database, the contents can be queried in the form of rows and columns using a cursor. Most of the system-provided content providers, such as the one for contacts, require a permission to access the data represented by the content provider. However, this is not always the case, although a valid content URI is needed to access a content provider. These URIs start with the prefix `content://`. To get the content URIs of the system-provided content providers, we analyse each queried field and each returned string value during method invocation. If a string starts with the prefix, the URI gets stored for later retrieval. Once the method invocations and field evaluations are done, the content of all detected and accessible content providers is dumped by iterating over all rows and columns. Similar to return values and fields, the dumped data is stored for later analysis.

3.3 Evaluation Phase

To detect possibly fingerprintable information, data from multiple devices have been collected. This data is analysed in the evaluation phase. In order to compare the data gathered from different devices, it is cleaned prior to comparison, as elaborated on in the following section.

3.3.1 Data Cleaning. By predefining multiple values or constants for a single parameter, methods can be invoked multiple times. Likewise, by calling methods of returned objects, a method can be called several times on different objects. Therefore, a single method or field can occur multiple times in the list of results. These occurrences are condensed in the cleaning step. In case the return values of these invocations do not differ, the different occurrences will be unified. Otherwise, the differing values will be kept for the analysis step.

On each device all obtainable information is collected twice. Between the two collection processes, the application is removed

completely. Afterwards, it is reinstalled using a different signing certificate to start the second collection. We use a different signing certificate to emulate a second application using an identical tracking library trying to detect the user. This allows us to eliminate all information sources which differ between application installs, such as the user ID or the data directory of the application. In addition, this measure makes it possible to exclude any information sources which change with application restarts, such as thread and process IDs, process times, or temporary native instance identifiers. However, some of these removed sources may in fact provide fingerprintable information. For example, the elapsed process time could be used to deduce the speed of the device. However, interpreting all of the changing values requires deeper knowledge about the peculiarities of each of these values. Therefore, we consider them out of the scope of this paper. Thus, these will be removed from the list of results.

3.3.2 Analysis. In the analysis step, the previously cleaned data of different devices is compared. The analysis framework discards all information sources which report the same value across all of the evaluated devices. If a method or field only exists on a subset of the tested devices, the reported values will only be compared across these devices. All of the information sources that differ between devices are marked as potentially fingerprintable. The list of fingerprintable data sources is finally sorted by class. For the following evaluation, this list is then categorised for better comparability.

4 EVALUATION

The following section discusses the coverage of constants, methods, fields, and content providers. The subsequent section details the three different test setups. Finally, we show the obtained results across the test setups.

4.1 Coverage

Given our approach is fully automated, API coverage is an important metric to assess its efficacy. We therefore provide coverage figures on constants (Section 4.1.1), methods (Section 4.1.2), fields (Section 4.1.3), and content providers (Section 4.1.4).

4.1.1 Constants. In the preparation phase 15791 possible constants were parsed from the Android API documentation. 4695 of these values were relevant, i.e., these values are either used in constructors or are used as parameters for methods which are invoked by the framework. Because constants are not clearly recognizable in the documentation, statements that are in fact not constants were also parsed. This can include, for example, variable names or types, such as *value* or *Integer*. These are included as they are also written in code blocks or are parts of hyperlinks. Furthermore, these non-interpretable statements can also include values which contain one or multiple dots. This includes for example sentence fragments, such as *i.e.* or placeholders such as *R.array.foo*. All of these occurrences are sorted out in the preparation phase after the interpretation fails. 1425 of the 4695 values were discarded due to this. Therefore, 3270 values were interpreted successfully and are thus usable to invoke constructors or call methods.

Table 1: Overview of method coverage.

	#	%
Documented Methods	39803	
Relevant Documented Methods	13643	
− Corresponding Class Not Found	250	
+ Additional Undocumented Public Methods	1388	
= Declared Public Methods	**14781**	**100%**
− Methods in Interface or Abstract Class	3296	22.30%
− Missed Methods Due to No Object	5433	36.76%
− Exception During Invocation	936	6.33%
− Blacklisted	11	0.07%
= Theoretically Invoked	5105	34.54%
Actually Invoked	**7528**	**50.93%**

Table 2: Overview of content provider coverage.

Declared Content Provider URIs	175
− Content Provider not Accessible	1
− Permission Required	136
− Other Exception	3
= Successfully Dumped	**35**

4.1.2 Methods. Table 1 shows an overview of the method coverage. In total, 13643 relevant methods are referenced in the API documentation. This includes some methods which are not available on the test device, for example, due to being deprecated. In total, 250 methods of 48 classes were not found on the device. However, an additional 1388 public methods exist on the device which are not documented. Thus, 14781 relevant and public methods are available on the test device. 3296 of the retrieved methods are in interfaces or abstract classes and can therefore not be created directly. For 5433 methods the corresponding object could not be created via a constructor. This could be, for example, due to missing parameters or other exceptions. 936 methods threw an exception during invocation. This can be due to missing permissions or, similar to constructors, invalid or missing parameters. 11 methods have been blacklisted due to side effects. This encompasses, for example, the method *java.util.concurrent.CompletableFuture.get* as it stopped the execution of the application. Thus, 5105 methods could be invoked by directly creating an object via the corresponding constructor. By exploring the non-primitive return values another 2423 methods were invoked. Therefore, in total, 7528 or 50.93% of the documented methods could be called.

4.1.3 Fields. In total, our approach discovered 26495 fields in the Android API. 24093 of these fields were static fields and did not require any object to be fetched. Thus, all of them could be retrieved. Of the remaining 2402 non-static fields, we were able to query 1151. For the remaining 1251 fields, no appropriate object to retrieve the fields value could be created. 1414 of the 26495 fields were private fields. Unlike private methods, fields which have only private visibility can still be fetched via Reflection. 285 of the 26495 fields were uninitialized, i.e., the retrieved value was *null*.

4.1.4 Content Providers. The overview of the content provider coverage are depicted in Table 2. In total, 175 content provider URIs have been found in fields and return values of methods. Of these 175 URIs, one was not accessible as it was a hidden API. 136 of the callable content providers required a permission while three threw an exception. Two of these exceptions were SQL exceptions due to non-existing databases, while one was a null pointer exception which occurred while querying the content provider. Thus, in total, 35 content providers could be queried.

4.2 Test Setups

We conducted our experiments on multiple devices to detect possibly fingerprintable information. Our main test devices have been two identical Google Pixel 4 devices with 64 GB of memory. Both have been updated to the latest software version as time of writing. The devices are running Android 10 with security patches from March 5 2020. In our first experiment, we evaluate the two devices for already existing possibly identifiable information. The devices have been set up in an identical fashion. Therefore, this setup represents the minimal set of identifiable information. In reality, after finishing the setup wizard, the device could already have different networks or display sizes configured and additional applications installed. In our second experiment, we customise one of the devices and rescan this device for new fingerprintable information. This setup of identical devices but varying user configuration allows us to detect methods, fields and content providers which can be used to fingerprint a user. In addition, we also compare two Samsung Galaxy S10e devices running Android 10 to identify if vendor customisations provide additional fingerprintable information. Finally, we also compare results from different devices and vendors. Methods, fields, and content providers which have been reported in one setup will not be considered again in the following setups.

The application gathers all data from sources which either require no permission or a normal permission. According to the API documentation, normal permissions protect resources which pose "very little risk to the user's privacy" [2]. Thus, we check whether this holds true in terms of fingerprinting. Wherever a normal permission is needed, the number of information sources requiring a normal permission are annotated in parentheses. However, the vast majority of information sources in our findings do not require a permission at all.

4.3 Results

The following sections show the results of the conducted experiments using the aforementioned test setups.

4.3.1 Fresh Setup. On new devices, only a small number of methods provide potentially fingerprintable information. Our framework detected 14 methods and one content provider which returned information that was differing between the devices. An overview of these information sources can be found in Table 3. These include information about the installed packages, free storage space, and accessibility services. Furthermore, the devices index the preinstalled ringtones during the first boot. The indexed information is available via the `content://media/internal/audio/media` content provider. In addition to various information about ringtones, the content provider also allows to query the timestamp of when the ringtone was added. Thus, if the time is set correctly on first boot (for example, after a factory reset), it is possible to query the time of device setup to the second.

Table 3: Overview of identified methods on a fresh setup. (M = methods, F = fields, CP = content providers)

Type	#M	#F	#CP
Unstable			
Media Timestamps	3	-	-
Native Instance IDs	-	2	-
User Start- & Unlocktime	2	-	-
Input Method	1	-	-
Uncategorised	-	1	-
PPID	1	-	-
Applications	1	-	-
Sum	8	3	0
Stable			
Applications	7	-	-
Available Storage	5	-	-
Accessibility Services	2	-	-
Ringtones (Timestamp)	-	-	1
Sum	14	0	1

In addition to the previous information sources, there are some less stable sources which survive an application reinstallation but change on device restart. As smartphones can be running for days or even weeks, such information could still be used to fingerprint a user. Eight methods and three fields provide such information. This includes information from the media player, such as current position or duration, the calling parent process id, and native instance identifiers of the used typeface and thread class. Furthermore, the descriptor of the input device, the version of the media store, and user unlock and start times also present unstable identifiers. While new devices do not provide a plethora of potentially fingerprintable information, smartphones usually get customised by the user to fit their needs. Thus, in the next step we will compare customised devices.

4.3.2 Customised Setups. In this setup one of the test devices was customised by altering settings and installing applications. 511 methods, 100 fields, and 181 rows in content providers yield differing values. The classification of these information sources can be found in Table 4. In addition to the methods and content provider found in the previous experiment, these allow to infer various settings and customisations done to the device. The most noticeable change in the results were achieved by altering the display and font size of various elements. The detected information sources report values such as the display metrics, default sizes of various UI components, or allocation sizes of bitmaps.

Another noticeable change that affected a lot of methods was altering the system language. While some methods and fields report the system language directly, others can be used to infer the language. This for example includes methods which report names of storage volumes such as *Internal Storage* vs *Interner Speicher* or button labels such as *On* vs *An*. Furthermore, changing the system locale also affected many methods, fields and content provider.

Table 4: Overview of identified information sources on a customised setup. The numbers in parentheses represent the share of methods which require a normal permission. (M = methods, F = fields, CP = content providers)

Type	#M	#F	#CP
Unstable			
Clipboard Content	13	-	-
Boot Count	-	-	2
Uncategorised	-	1	-
Sum	13	1	2
Stable			
Element Display Size	88	39	2
Additional Settings (Exist only if modified)	-	-	109
Language	105	3	-
Locale	62	2	1
Mobile & WiFi Network Details	44 (15)	10	4
Applications	13	27	-
Carrier, SIM	29	2	6
Enabled Radios (Mobile Data, NFC, WiFi, Location)	23 (19)	-	2
Accessibility Settings	6	1	17
Currency	22	-	-
Input Methods	16	1	3
Bluetooth Settings & Devices	17 (16)	-	1
Timezone	13	3	2
Uncategorised	11	3	1
Display Settings (Night Mode, Automatic Rotation, Refresh Rate, ...)	5	1	8
Colours	9	3	-
Wallpaper Settings & Colours	12	-	-
Caption Settings	3	2	4
Ringer Mode, Do Not Disturb	4	-	4
Default Apps (Dialer, Messages, Launcher)	6	-	1
Ringtone (Currently Set, List of Added Ringtones (incl. Timestamp))	5	1	-
Sound Effects	-	-	6
Audio Volume	5	-	-
Font Size	2	1	-
Device Name	1 (1)	-	2
Device Lock	3 (1)	-	-
Time & Date Format	2	1	-
Widgets	2	-	-
Various Timestamps	-	-	2
Notification Settings	-	-	2
Password Complexity	1 (1)	-	-
Personalization State	-	-	1
Screensaver	-	-	1
Activated Asisstant	-	-	1
User Restrictions	1	-	-
Settings Tiles (Order, Activated)	-	-	1
Security	1	-	-
Sum	511	100	181

Various accessibility settings can be queried using the Android API. It is, for example, possible to detect customised reaction timeouts, disabled animations, key repeat settings, or an adapted long press timeout. Using the captions API, it is possible to query the font size, colours, and locale specifically configured for captions. Furthermore, the device name, if customised, could uniquely identify a user. Interestingly, accessing the device name via the method *android.bluetooth.BluetoothAdapter.getName()* requires a normal permission, while it can be accessed without a permission via content providers. The Android API also allows to query UUIDs of bonded Bluetooth devices if a normal permission has been requested in the manifest. The `content://media/external/audio/media` content provider allows to query any custom ringtones added to the "Ringtones" directory. It allows to query detailed information about the added ringtones, such as title, duration, date added, date modified, file name, or file size. In contrast, querying added music files via the same content provider requires a dangerous permission.

The AudioManager API provides access to the audio volume set by the user. While the main volume may be changed more frequently, the API also allows access to other streams, such as ringer, notification, or alarm volume. Additionally, security APIs such as the KeyStore or BiometricManager allow to detect whether or not a user has setup a device lock or enrolled a fingerprint or other biometrics. Querying the biometrics requires a normal permission.

The content providers contain some entries only if they have been modified by the user or applications on the device. Thus, the customised device contains 109 additional entries which do not exist on the unmodified device. These include, for example, backup and night display settings, service URLs, and certificate blacklists.

Additional information sources which have been detected are the list of installed widgets and applications. These include detailed information about the applications, such as user ID, resource directories, and last update time. Changes to default applications for certain tasks, such as messaging or the dialler can also be queried. The activated launcher can be queried, for example, by supported features or icon sizes. AndroPRINT also detected some unstable information sources, which were changing between device restarts. These include methods reporting the current clipboard content or the boot count as reported by a content provider.

In addition to the two Pixel devices, we also analysed two Samsung Galaxy S10e devices using the Samsung Remote Test Lab[2]. Both devices are running Android 10. The devices have not been specifically customised by us, however, they still provided valuable identifiers. The additional identified information sources are depicted in Table 5.

In contrast to the Pixel devices, these two Samsung devices provide unique identifiers via different settings providers. The devices allow querying the e-mail address used to sign up for the Samsung account. Furthermore, any application has access to a user ID which stays the same across devices of the same user, a device-specific ID, an access token, and a smart tethering GUID. Using our Samsung Galaxy S9 running Android 10 with March 1 2020 security updates we were able to confirm these findings. As this device has an active SIM card, further features could be activated which provide additional unique identifiers. If the "Call & text on other devices"

[2]https://developer.samsung.com/remote-test-lab

Table 5: Overview of identified methods on Samsung devices. (M = methods, F = fields, CP = content providers)

Type	#M	#F	#CP
Samsung Specific Settings	-	-	26
Smart Tethering (Account Specific IDs, AES Keys, Activation Status, Family Size)	-	-	13
Timestamps (Security Policy Check Time, Settings Activation Times, Lockscreen Wallpaper Switch Time,...)	-	-	12
ToS Agreement (Status, Timestamp of Agreement)	-	-	8
Locale	1	-	7
Device & User ID, Connected Devices	-	-	4
Ringtone	1	-	3
Sound Effects	-	-	4
WiFi MWIPS Special SSIDs	-	-	4
Software Versions	-	-	3
Samsung Account Mail Address	-	-	2
WiFi Learning Score	-	-	2
Accessibility Settings	-	-	2
Logging Counts	-	-	2
WiFi P2P Device Name	-	-	1
Access Token	-	-	1
Uncategorised	-	-	1
SIP Address	-	-	1
Colours	-	-	1
Resource IDs	-	-	1
Theme Package	-	-	1
Sum	2	0	99

feature is activated, a SIP address can be queried which consists of the device ID and the access token. Additionally, the device ID and the user-settable device name of all devices connected via the “Call & text on other devices" are accessible. If the smart tethering feature is activated across family members, additional IDs can be queried. Furthermore, AES keys used for this smart tethering service can be retrieved. This `smart_tethering_AES_keys` entry contains what seem to be five 256bit AES keys and five timestamps. It seems that each AES key is valid for one month, as the five timestamps are all set in the future, each one month apart. After identifying all these unique identifiers, we have contacted the Samsung Mobile Security team on April 8, 2020 to disclose our findings responsibly. The findings have been acknowledged as a high severity vulnerability and are being fixed by Samsung.

In addition to the detected identifiers, AndroPRINT found multiple Samsung specific settings, such as edge or game settings, which differed between the two devices. Other differing values include different timestamps, such as feature activation times down to the second, agreement status to various Terms of Service, and software versions of proprietary services.

4.3.3 Cross-Device Setup. In the cross device setup, the results of different devices from various vendors are compared. In total AndroPRINT detected 193 additional methods, 154 fields, and 177

Table 6: Overview of identified methods on a cross-device setup. The numbers in parentheses represent the share of methods which require a normal permission. (M = methods, F = fields, CP = content providers)

Type	#M	#F	#CP
Resource IDs	14	101	-
Vendor Specific Settings	1	-	68
Display Size	23	5	3
Radio Features	13 (1)	5	9
Implementation Differences (View Instances, DownloadManager Database Structure,...)	15	7	3
Build Infos	2	20	1
Screen Characteristics	16	2	2
Volume, Audio Effects	3	2	14
Uncategorised	-	1	16
Applications	5	4	5
Supported Locales & Timezones	13	-	-
Input Devices (Hardware, Features, IDs)	9	-	3
Audio Support	11	-	-
CP DB Structure	-	-	10
Sounds	-	-	10
Java Runtime Differences	9 (2)	-	-
Media Codec Support	9	-	-
URLs	1	-	7
Developer Settings	-	-	8
MultiSIM	7	-	-
Storage Size (Internal, External, Cache)	6	-	-
Default Colours	4	-	2
Bluetooth Device Settings (Contain MAC Address)	-	-	5
Hardware Features	5	-	-
Camera Details	2	3	-
Screensaver Settings	-	-	4
Available Permissions	2	2	-
Notifications	-	-	4
Supported Gestures	-	-	3
Additional Hardware Support	3	-	-
RAM, Memory Class	3	-	-
Device Policy	2	-	-
Encryption Status	2	-	-
Default Dialer	2	-	-
Multiple User Support	2	-	-
GPS Hardware Details	2	-	-
MMS User Agent	2	-	-
Mount Paths	-	2	-
Admin Applications	1	-	-
Location Background Throttling Whitelist	1	-	-
Security Providers	1	-	-
System Fonts	1	-	-
DRM Engines	1	-	-
Sum	193	154	177

values in content providers which allow to detect device vendors and models. In particular, various resource IDs differed between the devices. Another category of values which differed between devices are vendor specific settings, such as gesture, pocket detection, or custom LED indicator settings. Similar to the previous experiment, the display size and resolution are again a distinctive feature between devices. AndroPRINT detected information sources which in particular report physical characteristics of the display, such as DPI values or physical dimensions. In addition to the dimension of the display, the supported features also differ between devices. This includes information sources which report, for example, refresh rates, HDR, or wide colour gamut support. The test devices also differ in the number and types of supported locales and the version of supported time zones, even on devices having the same security patch.

Another category of information sources which report different values across devices are information sources reporting hardware features. Using the different storage APIs it is possible to query the size of all storage volumes and partitions. Furthermore, the main memory of the device can also be queried using different methods. Additional hardware features which affect the reported values include radio features, multi SIM support, the number of available cameras, the camcorder quality profile, and support for features such as hearing aids or WiFi display. Additionally, it is possible to query the default dialler, differences in the audio handling, and security features such as encryption types, or device ID attestation support.

Some of the encountered differences stem from implementation and runtime differences. These categories consists of differences in the number of view instances, differences in database structures, varying line numbers, or buffer sizes.

Some of the devices, such as the Pixel 2 or the Galaxy S9, contain settings which comprise the MAC address of connected Bluetooth devices. From our testing, these entries are not created for newly added Bluetooth devices. Thus, we assume that they have been created while these devices were running Android 9, but have not been cleaned up with the Android 10 update. These entries seem to be persistent, as unpairing the affected Bluetooth devices does not remove them. Therefore, querying (previously) connected Bluetooth devices is possible even on Android 10, partly bypassing the `BLUETOOTH` normal permission usually required to query bonded devices.

In addition to differing return values or fields, the mere existence of additional fields or methods can also identify vendors or device models. For example, the Samsung devices in our evaluation report a number of proprietary fields. These are marked with the prefix *SEM* and are used to enable proprietary features. On the Samsung Galaxy S9 running Android 10 with March 1 2020 security updates, AndroPRINT was able to find 559 such additional fields. Furthermore, our framework detected differences in the structure of some content providers. For example, the settings content providers on Samsung devices report which application is responsible for a certain value, while Google devices do not.

5 DISCUSSION

As our study confirms, Google has removed access to many unique device identifiers with Android 10. However, the Android API still offers a lot of information sources which can be combined to create a fingerprint. Furthermore, our framework detected novel unique identifiers revealing new threats in terms of fingerprinting: While the core Android API offers ways to fingerprint users, which allows for identifying individuals, it does not directly leak personal information. As the case of Samsung shows, it is worth analysing vendor customisations, since these can potentially reveal personal information. Being able to silently query a user's email address effectively defeats the tight permission system of modern smartphone operating systems. This finding was unexpected, as it has far-fetching consequences and therefore requires a broader investigation (see below).

Due to the automatic nature of our approach AndroPRINT was able to find more fingerprintable information sources than previously identified. Our evaluation focused on Android 10, due to the privacy improvements added. Thus, our framework was not able to detect differences which only occur between different Android versions, such as version or API level differences. Unique identifiers, as identified by Torres and Jonker [16], have subsequently been removed from Android [3]. Our study was able to find new undocumented identifiers on certain Android devices. We searched for fingerprintable information sources which require no user consent to be obtained, i.e., our study mainly detected sources which do not need a permission or only a normal permission. Thus, all of the identified methods, fields, and content provider can be queried without user consent.

5.1 Limitations & Future Work

Our approach covers only a limited number of devices and customisations. Therefore, the numbers we present are only a lower estimate of the quantity of fingerprintable methods, fields, and content providers in Android 10. As our study has shown, certain vendor customisations are a particular threat in terms of fingerprinting. In order to fully quantify the consequences of vendor customisations with respect to fingerprinting and privacy, additional studies need to be performed. Therefore, we plan to extend our research to cover devices from other manufactures as well to try and compare the impact of different vendors customisations.

Even though we parse and query all defined constants in the Android API and predefine a number of different parameters, it is not possible to invoke all methods and create all objects. Thus, not all methods in the Android API could be invoked and neither all fields queried. Due to time and memory constraints, we also had to limit the number of objects which could be explored for further methods and fields. While the fingerprintable fields and return values of methods had different values in our experiments, our study does not assess the stability and potential diversity of these values over a longer period of time.

In the cleaning phase, values which differ between two identical method calls or field evaluations are removed. As argued in Section 3.3.1, some of these values could still be used to fingerprint a user given the meaning of these values is known. To deduce the meaning

of these values without requiring extensive manual labour machine learning could potentially be used.

The classification is based on a conservative approach. This means that information sources which have been classified to detect device and vendor differences may in fact also be modifiable by a user.

5.2 Countermeasures

Several browser extensions have tried to limit the fingerprintability of a browser by changing fingerprintable information sources, such as the user agent. However, our analysis has shown that similar information, such as the model or manufacturer of a device, can be accessed via a plethora of implicit and explicit methods. Similar to browser fingerprinting [4, 12], changing single methods, content provider entries, or fields, such as the well known device information in the *android.os.Build* class, may actually increase the fingerprintability of a device.

As our study shows, a lot of the customisations done to a device can be queried via the different settings content providers. Thus, we propose to limit the access to these content providers using a permission. As content providers already have the ability to regulate access, this measure could be implemented using these existing provisions. However, as our research has shown, a lot of the user settings can also be accessed or inferred via other methods and fields. Therefore, regulating these content providers is not a silver bullet to stop fingerprinting. Nevertheless, we hope that beyond limiting access to these content providers, AndroPRINT can help finding methods and fields which should be restricted.

6 CONCLUSIONS

In this paper we presented AndroPRINT, which is a novel framework designed to find fingerprintable methods, fields, and native content providers on Android. AndroPRINT automates the arduous task of searching for fingerprintable information sources. While some information sources have been removed over time or now require permissions to be accessed, there is still a vast amount of methods, fields, and content providers that allow accessing or inferring information about the user. This includes vendor and device-specific information, operating system specific information, and information that allows to infer user settings. Our automated approach allowed us to go beyond what manual analysis can detect. By dumping all available data we detected undocumented vendor customisations which provide unique device and user identifiers and even access to the users e-mail address.

Our approach can be used precautionary, i.e., one does not have to wait until information sources are used in the wild for fingerprinting. Therefore, AndroPRINT makes it possible to detect fingerprintable information sources before these can be used to track users. When used in the development process of new or modified Android versions AndroPRINT can reveal leaking information sources before these can be abused for fingerprinting.

REFERENCES

[1] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *Network and Distributed System Security Symposium – NDSS 2017*. The Internet Society.

[2] Android Developers. 2019. Permissions Overview - Protection Levels. https://developer.android.com/guide/topics/permissions/overview/#normal-dangerous

[3] Android Developers. 2019. Privacy Changes in Android 10. https://developer.android.com/about/versions/10/privacy/

[4] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *Privacy Enhancing Technologies – PET 2010 (LNCS)*, Vol. 6205. Springer, 1–18.

[5] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. 2015. FPGuard: Detection and Prevention of Browser Fingerprinting. In *Data and Applications Security and Privacy – DBSec 2015 (LNCS)*, Vol. 9149. Springer, 293–308.

[6] Fyber and Sapio Research. 2019. 2019 State of In-App Advertising and Monetization. https://www.statista.com/statistics/262945/revenue-development-of-mobile-apps

[7] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: automatically generating heuristics to detect Android emulators. In *Annual Computer Security Applications Conference – ACSAC 2014*. ACM, 216–225.

[8] Andreas Kurtz, Hugo Gascon, Tobias Becker, Konrad Rieck, and Felix C. Freiling. 2016. Fingerprinting Mobile Devices Using Personalized Configurations. *PoPETs* 2016 (2016), 4–19.

[9] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. 2015. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. ACM, 820–830.

[10] Mike Perry, Erinn Clark, Steven Murdoch, and Georg Koppen. 2019. The Design and Implementation of the Tor Browser - Cross-Origin Fingerprinting Unlinkability. https://www.torproject.org/projects/torbrowser/design/#fingerprinting-linkability

[11] Iskander Sánchez-Rola, Igor Santos, and Davide Balzarotti. 2017. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Security Symposium 2017*. USENIX Association, 679–694.

[12] Michael Schwarz, Florian Lackner, and Daniel Gruss. 2019. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Network and Distributed System Security Symposium – NDSS 2019*. The Internet Society.

[13] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In *Asia Conference on Computer and Communications Security – AsiaCCS*. ACM, 749–763.

[14] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. 2018. SCAnDroid: Automated Side-Channel Analysis of Android APIs. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2018*. ACM, 224–235.

[15] Oleksii Starov and Nick Nikiforakis. 2017. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *IEEE Symposium on Security and Privacy – S&P 2017*. IEEE Computer Society, 941–956.

[16] Christof Ferreira Torres and Hugo Jonker. 2018. Investigating Fingerprinters and Fingerprinting-Alike Behaviour of Android Applications. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II (LNCS)*, Vol. 11099. Springer, 60–80.

[17] Christof Ferreira Torres, Hugo L. Jonker, and Sjouke Mauw. 2015. FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II (LNCS)*, Vol. 9327. Springer, 3–19.

[18] Marissa Wood. 2019. Today's Firefox Blocks Third-Party Tracking Cookies and Cryptomining by Default. https://blog.mozilla.org/blog/2019/09/03/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/

[19] Wenjia Wu, Jianan Wu, Yanhao Wang, Zhen Ling, and Ming Yang. 2016. Efficient Fingerprinting-Based Android Device Identification With Zero-Permission Identifiers. *IEEE Access* 4 (2016), 8073–8083.