EXADPrinter: Semi-Exhaustive Permissionless Device Fingerprinting Within the Android Ecosystem

Sihem Bouhenniche
Univ. Lille, CNRS, Inria
Lille, France
sihem.bouhenniche@univ-lille.fr

Pierre Laperdrix
Univ. Lille, CNRS, Inria
Lille, France
pierre.laperdrix@univ-lille.fr

Walter Rudametkin
Univ. Rennes, CNRS, Inria, IRISA, IUF
Rennes, France
walter.rudametkin@irisa.fr

Abstract

Android is the dominant mobile operating system, powering more than 70% of mobile devices and presenting a significant opportunity for user tracking. As privacy regulations tighten around how personal data can be used and collected, trackers are looking for alternatives that are under less scrutiny to evade detection. Device fingerprinting has emerged as a key solution, allowing trackers to create identifiers without user consent in a stealthy manner. Despite the extensive research on fingerprinting done from a web browser in the past decade, device fingerprinting on Android remains relatively understudied, with limited literature exploring its specific techniques and implications for user privacy.

In this study, we introduce EXADPrinter, a novel semi-exhaustive permissionless device fingerprinting framework targeting Android devices. Without requiring permissions, our framework extracts over 200,000 properties per device by leveraging methods such as Java reflection and execution of shell commands. Through a dedicated Android application and a 13-month data collection, we gathered over 4,004 fingerprints coming from 3,143 different Android devices, covering 68 manufacturers and 9 Android versions ranging from 8 to 16.

Through our framework, we demonstrate that diverse data can be collected about the hardware, the operating system and the user without requiring special permissions. We show that combining 5 attributes without any IDs or personal information is enough to identify 100% of devices of our dataset, painting a bleak picture of the current state of the Android ecosystem. Moreover, our framework highlights the negative impact of custom operating systems and manufacturer-specific customizations as they enhance the effectiveness of device fingerprinting. Furthermore, EXADPrinter uncovers some leakage of sensitive information caused essentially by manufacturer customizations, including the exposure of user emails, emergency contacts, and persistent identifiers such as SIM identifiers.

Keywords

Device Fingerprinting, Tracking, Personally Identifiable Information, Android

Proceedings on Privacy Enhancing Technologies 2026(3), 278–295

https://doi.org/10.56553/popets-2026-0082

1 Introduction

As of November 2025, Android's global market share for mobile operating systems is estimated to be 71.9% [40]. This popularity is favored by its open-source nature, which allows various actors, from device manufacturers to independent developers and communities, to customize the core framework known as the *Android Open Source Project* (AOSP) [8]. Moreover, the vast selection of Android applications available across different app marketplaces significantly contributes to its popularity. For example, Google Play Store, the largest marketplace for Android apps, hosted approximately 1.6 million applications as of November 2025 [10]. However, the majority of Android applications include at least one tracker, used for different purposes like analytics, user profiling, and targeted advertising [27, 41]. These trackers often operate without the user's knowledge, discreetly collecting data about both the device and the user, and frequently gathering more information than is necessary for their functionality, which raises significant security and privacy concerns.

Tracking relies on identifying either the user, the device, or both. In earlier versions of Android, third-party applications had access to persistent hardware-based identifiers such as the *International Mobile Equipment Identity* (IMEI/MEID), SIM card-related identifiers like the *International Mobile Subscriber Identity* (IMSI), and network-based identifiers such as the WiFi and Bluetooth *Media Access Control* (MAC) addresses. To enhance user privacy, access to these identifiers has been progressively restricted by the Android Permission System. Starting from Android 10, access to certain sensitive identifiers was limited to system applications holding the system permission `READ_PRIVILEGED_PHONE_STATE` [9]. To mitigate the use of persistent identifiers, Android introduced *Android Identifier* (ANDROID_ID) [7], a unique identifier for each <*app-signing key*, *user*, *device*> combination. Additionally, Google Play Services provides its own *Advertising Identifier* (AD_ID) [21], an identifier specifically designed for advertising purposes. Developers can use it after requesting the `com.google.android.gms.permission.AD_ID` permission. It is important to note that unlike persistent identifiers, these can be reset or deleted by users. As a result, trackers now employ a combination of techniques to achieve more precise and stable device identification [22].

Device fingerprinting serves as an alternative to identifier-based tracking. By aggregating specific hardware and software attributes, trackers can synthesize a unique signature that distinguishes a specific device. This technique exploits the high *entropy* of modern computing environments; the vast variations in hardware components, Operating System (OS) versions, and user configurations result in unique combinations that identify devices.

In academia, fingerprinting has mostly focused on Web browsers by collecting information such as device models, OSes or timezones via calls to the browser's APIs [16, 26, 42]. The literature on device fingerprinting, which goes beyond just using the browser for data collection, is much thinner. Previous studies on Android device fingerprinting [18, 23, 43] have employed manual methods relying on small, manually selected, sets of attributes. A more recent study in 2020 [31] is broader in terms of attribute collection but is tested on a small number of devices, which fails to reflect the diversity of the Android ecosystem, which encompasses a large number of manufacturers, device models, and OS versions.

Device fingerprinting is a statistical attack for re-identification. The limitations of the state-of-the-art underscore the need for broader investigations into the privacy risks associated, in particular, with fingerprinting of Android devices.

In the industry, the perception of this technique from platform holders has shifted. On Apple's side, fingerprinting of iOS devices is forbidden. The Apple Developer Program License Agreement that every developer must sign to publish their app on the app store [11] states that "*neither You nor Your Application will derive any data from a device for the purpose of uniquely identifying it or a user*". Apple has shown in the past that applications that fingerprint are not welcome, with several bans in 2021 of applications using the China Advertising ID (CAID) that relied on device fingerprinting for tracking [32]. On Google's side, a 2019 article indicated that fingerprinting "*subverts user choice and is wrong*" and that they planned to "*more aggressively block fingerprinting*" [38]. However, in January 2025, Google announced that they were dropping their prohibition of fingerprinting in the Ads platform program policies due to "*advances to privacy-enhancing technologies*" and "*the rise of new ad-supported devices and platforms*" [20]. This prompted the Information Commissioner's Office (ICO), the UK's data watchdog, to state that this change is "*irresponsible*" and that it will be "*a high bar to meet*" for organizations to demonstrate that their use of the technique complies with the requirements of data protection law [2].

With recent policy changes from Google coupled with the limited literature on the subject, our goal with this study is to advance our understanding of device fingerprinting on Android and characterize its various aspects in depth. Notably, we want to understand the level of threat device fingerprinting poses to user privacy.

To achieve this, we introduce **EXADPrinter**, a novel semi-exhaustive permissionless fingerprinting framework for Android devices that automatically extracts device attributes using multiple extraction techniques, such as Java Reflection, the execution of shell commands and retrieving content provider values. Our framework allows us to collect large fingerprints that include more than 200,000 attributes and works across Android manufacturers, devices and versions.

The main contributions of this study are:

1) Semi-Exhaustive and Scalable Fingerprinting Framework. We introduce a semi-exhaustive fingerprinting framework that automatically extracts device attributes, which makes it scalable across different Android versions and manufacturers since it does not rely on a predefined set of attributes and instead dynamically discovers them through attribute exploration.

2) Comprehensive Fingerprint Dataset. We collected 4,004 fingerprints from 3,143 devices using a fingerprinting app published in the Google Play Store. The dataset spans 9 Android versions and 68 manufacturers, with each fingerprint containing more than 200,000 attributes.

3) Fingerprint Attribute Analysis. We conduct an in-depth analysis of fingerprint attributes by measuring their individual uniqueness and entropy. Our study highlights the impact of user, operating system, and manufacturer customizations on fingerprint uniqueness, and shows how these factors contribute significantly to the creation of unique fingerprints.

4) Fingerprintability and Stability Evaluation. We show that we can uniquely identify 100% of devices in our dataset by using only **five** attributes. Additionally, we assess if long-term tracking is possible by keeping only the most stable attributes that do not change over time on all devices. We achieve 99.62% of fingerprint uniqueness with a combination of **28 stable** attributes, showing that fingerprinting can be effective over a longer period of time.

Furthermore, our comprehensive dataset allowed us to discover several attributes that compromise user privacy by leaking sensitive information. In particular, we identified attributes exposing eight distinct types of sensitive data, including *Personally Identifiable Information* (PII), such as email addresses, and persistent identifiers, such as SIM card IDs. In total, 25.96% of the devices in our dataset contain at least one such identifier.

2 Background and Related Works

Android is an open-source mobile OS that allows both *Original Equipment Manufacturer*s (OEMs) and developers to customize and specialize it. It is available through *Android Open Source Project* (AOSP) [8], a Google-led project that provides the core framework, source code, and documentation necessary to build Android variants. In this section, we provide an overview of key concepts related to user identification within the Android ecosystem and introduce the fingerprinting techniques commonly recognized in the research community.

2.1 Android Device Identifiers

Android identifiers can be categorized into two main categories:

1) Persistent Identifiers. Permanent identifiers that are difficult to change, even after a factory reset of the device. According to the official Android documentation [9] and several prior studies [29, 33, 35], the most common persistent identifiers include the device *serial number*, which is a unique identifier assigned by the manufacturer to identify an individual device, and the *International Mobile Equipment Identity* (IMEI/MEID), used for communication devices. In addition, there are identifiers linked to the *Subscriber Identity Module* (SIM) card, such as the *International Mobile Subscriber Identity* (IMSI), which is a 64-bit number that uniquely identifies a cellular network subscriber, and the *Integrated Circuit Card Identifier* (ICCID), which uniquely identifies the SIM card itself. Moreover, persistent identifiers include also *WiFi* and *Bluetooth MAC* addresses.

2) Non-Persistent Identifiers. Non-Permanent identifiers that can be reset, changed, or deleted, such as the *ANDROID_ID* which is a 64-bit bit unique unique for each combination of app-signing key,

user, and device expressed as a hexadecimal string. Also, Google provides *AD_ID*, a unique, user-resettable, and user-deletable identifier used for advertising purposes. In addition, *Personally Identifiable Information*s (PIIs) such as email addresses and phone numbers can be used to identify devices.

Accessing these identifiers, or sensitive data in general, remains a significant area of research. Even with regular security updates, such as the restrictions introduced in Android 10 [9] and stricter laws like the *General Data Protection Regulation* (GDPR) [17], recent studies have introduced new techniques for bypassing existing security controls. For example, Reardon et al. [34] demonstrated how some apps can exploit Android's permission system to access private data without the user's informed consent, emphasizing the difficulty of fully enforcing runtime permissions. Meng et al. [29] examined security improvements made to Android after the introduction of GDPR but still identified vulnerabilities, such as the continued exposure of persistent identifiers such as MAC addresses, primarily due to customizations by manufacturers. Similarly, Dong et al. [15] uncovered covert third-party identifiers stored in external storage, presenting another method for apps to evade Android's tracking restrictions.

2.2 Android Device Fingerprinting

Device fingerprinting is the process of passively extracting a collection of distinctive properties, called **attributes**, from a device to generate a unique fingerprint. A fingerprint typically consists of hardware-related attributes (e.g., device manufacturer and model), software-related attributes (e.g., operating system version), and network-related attributes (e.g., HTTP user-agent header). Together, these attributes can serve as a unique identifier for the device.

In the literature, several studies have explored mobile device fingerprinting techniques [18, 23, 31, 43]. The majority of these techniques are manual and target specific attributes. For example, Hupperich et al. [23] investigated tracking techniques used for commercial purposes on mobile devices. They proposed a fingerprinting method by extracting four categories of attributes: browser, system, hardware, and behavioral attributes, using a Web application executed in a mobile browser.

In a different approach, Wu et al. [43] explored the identification of Android devices based on non-sensitive attributes (i.e., without requiring special permissions) accessible through the native Android API. They collected 38 attributes from 2,239 devices using a native application, filtered out irrelevant data, and applied a classification algorithm to achieve device identification. While their method demonstrated high accuracy at the time, it is now relatively outdated, as multiple Android versions have since been released, introducing significant changes to the operating system and its permission model.

Palfinger et al. [31] introduced an automated method for device fingerprinting by analyzing the Android API using Java Reflection [30] to explore available system attributes. As detailed in Section 3.2.2, our work goes beyond their approach by extending the list of selected methods for invocation and by improving the instantiation process of class objects. Moreover, their study was limited to a single Android version (Android 10) and was tested on only 4 devices from only 2 manufacturers. Our dataset provides a better representation of the diversity of the Android ecosystem with 3,143 devices from 68 manufacturers across 9 Android versions.

Kondracki et al. [24] used fingerprinting techniques to distinguish between regular users and Android sandboxes. The authors used a set of 81 manually collected attributes and focused primarily on behavioral differences and the fact that regular users exhibit more human-like usage patterns, such as taking photos, having contacts and a more diverse set of installed packages. Although their approach reported a detection accuracy of 98.54%, it relies heavily on *dangerous permissions*, which require explicit consent from users.

Specter et al. [39] statically analyzed over 400,000 Android *Software Development Kit*s (SDKs) and applications. Specifically, they reported the exfiltration of over 500 distinct device signals that can be used for fingerprinting. Their findings reveal that many SDKs exhibit fingerprinting behavior that is not clearly disclosed and that these fingerprinting-like SDKs are integrated within popular applications, highlighting the widespread use of fingerprinting behaviors in practice and their adoption in real-world mobile apps.

In our work, we focus on proposing a semi-exhaustive fingerprinting method that goes beyond existing work by collecting a significantly larger set of more than 200,000 attributes without requiring any special permissions. We combine several techniques to achieve this goal, where Java Reflection constitutes one component of a broader framework designed to maximize access to a more comprehensive set of device attributes.

3 EXADPrinter: Methodology

Our methodology consists of three steps. We begin with **Domain Exploration**, where we collect all documented classes available in the official Android Developer Documentation [6]. Next, in the **Attribute Extraction** step, we introduce a method for semi-exhaustively extracting device attributes using three techniques. Finally, in the last step, called **Device Identification**, we analyze fingerprints to identify the most relevant attributes for device identification. Figure 1 summarizes our approach.

3.1 Domain Exploration

This step collects the list of documented Android API classes that will be used to seed the attribute extraction process. We scraped the HTML pages of the official Android documentation website [6]. In total, we collected 6,022 documented classes that define 65,259 documented fields and 212,599 documented methods. The scraping included all documented API elements across all Android versions up to API level 35, the latest available as of September 2024.

Additionally, we conducted a preliminary study to compare the Android APIs officially documented by Google with those actually present on real devices from various manufacturers. To do this, we used the BrowserStack platform [13] to access real devices and applied Java Reflection to inspect the available classes and methods. We found that Java Reflection was powerful in uncovering hidden or undocumented SDK APIs, as well as manufacturer-specific APIs and it motivated us to adopt it for our final framework. Appendix A.1 provides additional information about our findings.

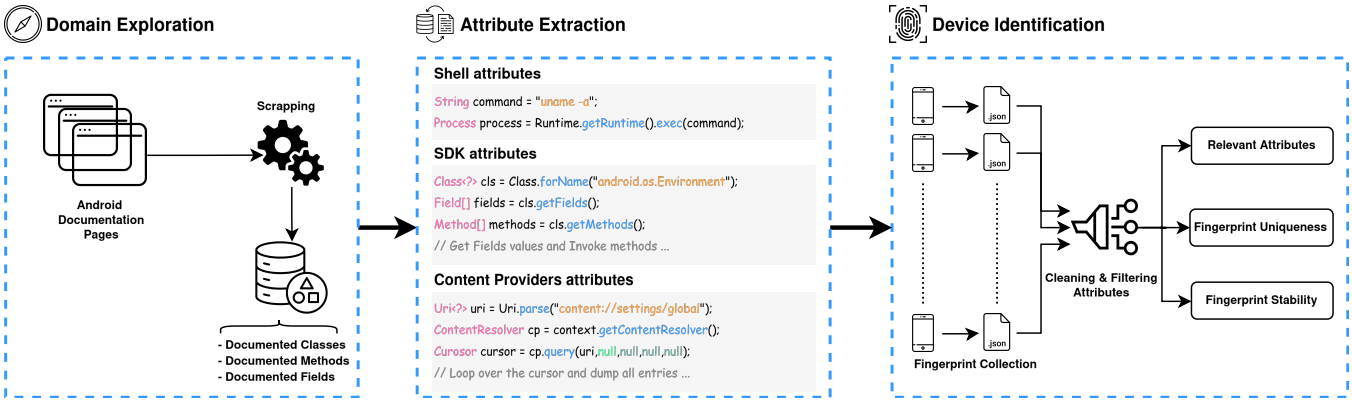


Figure 1: EXADPrinter methodology overview.

3.2 Attribute Extraction

This constitutes the main step of our methodology, where we introduce our semi-exhaustive approach to extract device attributes. For that, we combined three different extraction techniques: executing shell commands, exploring all Android-documented Java classes using Java Reflection, and retrieving content provider values. These three types of attributes together constitute the fingerprint.

3.2.1 Shell Attributes. Tools such as the Android Debug Bridge (ADB) [5], which gives you command line access, allow shell commands to be executed. Prior work [43] relied on a limited set of 5 shell commands to collect information on the device kernel or the list of fonts. We looked at the full set of over 200 commands provided by **Toybox** [25], which is a collection of command-line tools integrated into the AOSP since version 6.0 (Marshmallow). We looked at each command's relevance for fingerprinting while making sure it would not perform system modifications. We excluded commands that were deterministic with no system-unique output, like *echo* or *base64*, commands that required custom arguments, like *ping* or *file*, or commands that could alter the system, like *mount* or *kill*. We kept the 41 commands listed in Appendix A.2.
To run the commands, we use the `getRuntime().exec(cmd)` method of **Runtime** class[1].

3.2.2 SDK Attributes. Starting with the list of documented Android classes, we explore each class using the **Java Reflection API** (`java.lang.reflect`) [30] by enumerating its fields and their values, listing all methods and invoking them. Reflection allows inspecting and manipulating internal properties of the program at runtime. This step involves instantiating classes to get fields values or invoke methods, as well as creating and passing parameters to method calls. Figure 5 in Appendix A.3 summarizes the steps involved in this process.

Starting class list. Our aim with the initial list is to start from classes that are available on all Android devices, no matter the manufacturer. For that purpose, we relied on the documented APIs as they are stable across Android versions and developers rely on them when building their applications. To increase the starting list, we had considered reverse-engineering every Android OEM variant for the over 200 [36] manufacturers to capture undocumented classes. However, due to the extensive and complex nature of this effort, we focused solely on documented classes.

Class exploration. We attempt to load every documented class using the `Class.forName` method. If a class is loaded, we proceed to extract its attributes using `Class.getFields()` and methods using `Class.getMethods()`. This allows us to get the full list of public APIs including undocumented and custom ones. For every field, we check if it is static, in which case we can retrieve its value directly from the class. If not, we must instantiate its class. Similarly, we invoke every static method without parameters directly. If the field or the method is not static, and/or requires parameters, we create the necessary objects before invoking them. In case of errors when accessing field values or invoking methods, we catch the exception, record the error, and continue the extraction process.
For parameter creation, if it is a primitive type, we return a default value (e.g., 1 for integers, an empty string for strings). If it is an object, we instantiate the class as described below.
For class instantiation, we obtain instances of specific classes (i.e., `android.content.Context`, `android.content.pm.PackageManager`, `android.content.ContentResolver`). Next, we automatically retrieve all available system service managers by iterating over the service names defined in `Context` and calling the `getSystemService`[2] method, and we store these instances in a list that we use when querying fields or invoking methods of non-static classes. For other classes, we retrieve their constructors and sort them by the number and types of parameters. We prioritize constructors with fewer parameters and primitive types over object parameters. Then, we attempt to invoke each constructor in order; when it succeeds, we store the instance in the list.
When invoking methods, We do not invoke all methods of a given class. Instead, we select methods that perform reading operations without modifying the state of the object to avoid potential app crashes and unintended modifications to the system. Specifically, we focus on *getter* methods. To identify such methods, we only consider those starting with specific prefixes, such as `get`, `is`

[1] https://developer.android.com/reference/java/lang/Runtime

[2] https://developer.android.com/reference/android/content/Context#getSystemService(java.lang.String)

and has. Some prefixes are specific to certain vendors and were discovered during the domain exploration step, such as semGet for Samsung and oppoGet for OPPO. Listing 2 in Appendix A.3 provides the complete list of these prefixes. For the methods that we do not invoke, we still record their existence in order to capture the actual implementation of the class on the device, since vendors may add or remove methods.

3.2.3 Content Provider Attributes. A *Content Provider* [4] is an Android component that offers a structured way to store, manage, and control access to data identified by a unique *Uniform Resource Identifier* (URI). To grant secure access, Content Providers use the Android permission system. For example, to read from the Contacts Content Provider or to modify its content, applications must request the READ_CONTACTS or WRITE_CONTACTS permissions. By using such permissions, Content Providers allow sharing data between applications in a secure way. However, not all Content Providers are protected by permissions. Some of them provide read access without any permission. EXADPrinter retrieves all possible values from these Content Providers. To achieve this, we first collect every potential content provider URI when gathering values from fields by Java reflection. A potential URI is a string starting with "content://". After that, we query each Content Provider by its URI and try to retrieve any values.

3.3 Device Identification

The main objective of this step is to evaluate the importance of each attribute and its contribution to device uniqueness. After that, we select relevant attributes and assess the overall fingerprintability of our dataset by analyzing both the uniqueness and stability of the collected fingerprints.

3.3.1 Data Preparation & Cleaning. Given that our extraction method generates heterogeneous data (e.g., numbers, strings, lists, objects), we start with a normalization step. We convert complex data types to primitive types (numbers and strings); objects are flattened, and lists are sorted and converted to strings. The goal is to create a one-level JSON object that represents the fingerprint. Next, we clean our fingerprints by removing:
1) Attributes associated with execution exceptions (for which we saved the exception type); 2) Attributes representing methods that we did not invoke; 3) Unchanged attributes whose values remain identical across all fingerprints; and 4) Attributes that are globally unstable (they change too frequently to be used for device fingerprinting).

We define the **Number of distinct values** $D(A)$ of an attribute A as the number of different values it can have across all fingerprints:

$$\mathrm{D}(A) = |\{A(f) \mid f \in F\}| \tag{1}$$

Unchanged attributes are constant and always have a number of distinct values D(A) equal to 1. Since these attributes do not vary across devices, they do not contribute to the uniqueness of the fingerprint, making them irrelevant for fingerprinting purposes.

For devices having more than one fingerprint, we define the **intra-device change count** $C(A, d)$ of an attribute A for a device $d \in D_{all}$, as the number of value changes observed for A across all fingerprints $f \in F_d$ of that device d.

$$C(A, d) = \sum_{i=2}^{n} \mathbf{1}\left(v(A, f_i) \neq v(A, f_{i-1})\right) \tag{2}$$

An attribute is considered **stable** on a device d if its value never changes across its fingerprints ($C(A, d) = 0$).

We further define the **inter-device change count** $C_{\mathbf{inter}}(A)$ of an attribute A, as the number of devices for which $C(A, d) > 0$. This metric quantifies the **stability** of an attribute A across all devices.

$$C_{\mathrm{inter}}(A) = \sum_{d \in D_{all}} \mathbf{1}(C(A, d) > 0) \tag{3}$$

An attribute is considered **globally unstable** if it changes its value for every device in D_{all}.

3.3.2 Attribute Selection. To evaluate the importance of each attribute and its contribution to device fingerprinting, we use two metrics.
Number of unique values $U(A)$, which is defined as the number of values that appear exactly once across all collected fingerprints for an attribute A.
Formally, let $V_A = \{A(f) \mid f \in F\}$ be the set of all observed values for attribute A, the number of unique values $U(A)$ is then given by:

$$U(A) = |\{v \in V_A \mid |\{f \in F \mid A(f) = v\}| = 1\}| \tag{4}$$

where $|\{f \in F \mid A(f) = v\}|$ represents the frequency of occurrence of value v.
Shannon entropy, originally used to measure species diversity in ecological studies, quantifies the uncertainty or variability of a dataset. This metric is used in fingerprinting to measure the variability of fingerprints within a dataset [26]. The Entropy H for an attribute A with possible values $a1, a2, .., a_n$ is defined by the following formula:

$$H(A) = -\sum_{i=1}^{n} P(a_i) \log_2 P(a_i) \tag{5}$$

where $P(a_i)$ is the probability of occurrence of value a_i within the dataset, and n is the total number of distinct values the attribute can take.
In the context of fingerprinting, attributes with a higher number of unique values and entropy have greater diversity, making them more important. To compare attributes with varying value ranges, we use the normalized version of this entropy defined as:

$$H_{norm}(A) = \frac{H(A)}{H_{max}} \tag{6}$$

where $H_{max} = \log_2(n)$ represents the maximum entropy for an attribute with n distinct values.

4 Dataset

We collected our dataset over 13 months. In this section, we present the fingerprint collection process, provide an overview of the data and its diversity across Android versions and device manufacturers, and describe the structure of our device fingerprints.

4.1 Data Collection

To collect fingerprints from real Android devices, we developed a dedicated Android application that implements our extraction process described in Section 3.2. The application is used solely for device fingerprinting, and this purpose is clearly communicated to

users. On average, the extraction takes approximately one minute to complete.

Once extracted, the fingerprint is temporarily stored in the device's cache for five minutes to prevent duplicate captures within short time intervals. To track devices, we generate a random *Universally Unique Identifier* (UUID) for the fingerprint and save it as a `SharedPreferences` object[3].

To allow network transmission, our application uses the `INTERNET` permission[4]. This permission has a normal protection level and is automatically granted at install time. It is the only permission required by our application to run and the large majority of Android applications request it [1]. Our application is available on the Google Play Store at https://play.google.com/store/apps/details?id=com.amiunique.amiuniqueapp since October 2024.

4.2 Ethical considerations

We know that working on device fingerprinting is a sensitive topic as it has the potential to single out individual devices and, to some extent, the users behind them. Compared to desktops or laptops, smartphones are less likely to be shared. While developing and deploying our mobile application, we followed the best practices to minimize risks and protect user information. We describe this process in more detail below.

Institutional Authorization. We obtained an IRB approval from the institutional ethics board of our organization to conduct the experiments for this study.

Experimental environment. The application was developed internally in our lab and only people actively involved in the study had access to its code and, more importantly, to its database. All data from the application is encrypted for transit and we do not use any third-party services to collect more data or process what we have.

Informing users about data collection. Our application follows established guidelines for user consent in compliance with relevant privacy regulations, including the GDPR. The participants must first go to the Play Store where a description of the application can be found along with a link to our privacy policy. Inside the application, the participants are welcomed with additional details about what our application does along with a link to our full privacy policy. It is not until the user gives their consent, by checking the *"I consent to the collection of my device fingerprint"* checkbox and clicking the *"Scan my device fingerprint"* button, that our algorithm runs. Users can request access or deletion of their device fingerprints at any time.

Steps before deploying our application. Before deploying our application at scale, we conducted internal tests with members of our lab to assess that our application was working as intended. Surprisingly, we did not anticipate that we would collect email addresses nor access clipboard content without a single permission request. Especially, for the clipboard content, a deprecated method from the `ClipboardManager` class[5] was still working at the time of testing and provides access to its content silently, that is, without warning the user. Considering the sensitivity of this data and potential privacy concerns, we immediately blacklisted access to clipboard contents and replaced email addresses with an "`email address [hash]`" string before sending them to our server, ensuring that this data is never stored in our database.

Steps we took after deploying our application. Because of our internal tests, we decided to run tailored regular expressions periodically on our database to detect the potential collection of personal data, like phone numbers and addresses. Two expressions were triggered during our experiment and led to the discovery of emergency contacts on Samsung, Xiaomi phones. We updated our application to remove the collection of this data and went through all the collected fingerprints to remove the values of the affected attributes, similarly to what we did with the email addresses.

Disclosure to the affected parties. We disclosed the information leakage and vulnerabilities we found to the affected parties. For the sensitive information that was found coming from specific vendors, we use their associated bug bounty programs to report the leaks. We sent reports to Samsung, OnePlus, and Xiaomi for the problems we found that were affecting only their models. Samsung assigned a moderate severity rating to the reported issues and told us that a patch is currently being worked on to fix them. Xiaomi reported that the problems on their models were already fixed in the latest versions of their firmware. OnePlus acknowledged the reported issues and closed the ticket, stating that it was already a known security issue. For issues affecting multiple vendors, as well as our general findings related to the fingerprintability of the Android ecosystem, we reported them to Google. We were told that several attributes we identified were previously reported by an internal Google engineer and are awaiting a fix. Notably, the Bluetooth issues described in Section 5.2.1 will be fixed as part of a bigger revamp of the handling of MAC addresses on Android.

4.3 Dataset Overview

Our dataset includes **4,004 fingerprints** from **3,143 distinct devices** collected between October 2024 and November 2025, covering 9 Android versions from 8 to 16, which represents 91.7% of total Android usage [3]. Table 1 shows the distribution of fingerprints across Android versions.

Table 1: Distribution of devices across Android versions

Version	8	9	10	11	12		13	14	15	16
SDK Level	26	28	29	30	31	32	33	34	35	36
Fingerprints	1	87	147	185	253	7	572	970	1292	490
Devices	1	73	129	164	214	5	410	718	1030	427

In terms of manufacturer diversity, our fingerprints come from 68 different manufacturers. Figure 2 shows the top 10 manufacturers. Extracted fingerprints are stored in JSON object files, where each key corresponds to a fingerprint attribute. Each fingerprint object contains over 200,000 attributes, with the smallest containing 206,831 attributes and the largest 288,423. Listing 3 in Appendix A.4 shows a fingerprint snippet.

[3]https://developer.android.com/training/data-storage/shared-preferences
[4]https://developer.android.com/reference/android/Manifest.permission#INTERNET
[5]https://developer.android.com/reference/android/content/ClipboardManager

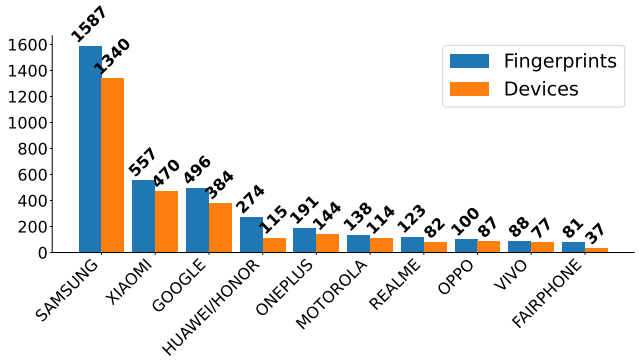


Figure 2: Top 10 Android manufacturers in our dataset.

4.4 Attribute Extraction Evaluation

Collected fingerprints are composed predominantly of SDK attributes, which represent over 99% of their total size, followed by content-provider attributes.

Shell attributes. We successfully executed an average of 28 (68.29%) commands per fingerprint out of the 41 shell commands that we attempted to execute. The encountered errors are caused by permission denials or commands being inaccessible or not found.

SDK attributes. We enumerated an average of 263,934 SDK attributes per fingerprint. Among these, we selected 41.31% of the methods for execution based on predefined prefixes. After method invocation using reflection, we observed that on average only 5,675 (15%) of invoked methods per fingerprint caused errors, with a maximum of 6,728 and a minimum of 3,453. The majority of these errors were `InvocationTargetException` (45.28%), `IllegalArgumentException` (35.42%), and `NullPointerException` (16.84%).

Content-provider attributes. We enumerated an average of 174 URIs. Among them, we were able to successfully query three content providers: *content://settings/global*, *content://settings/secure*, and *content://settings/system*. Together, these content providers contribute with an average of 1,101 attributes per fingerprint. The remaining content providers could not be accessed due to missing permissions.

5 Detected Unique Identifiers

The goal of this section is to explore our dataset to detect common unique identifiers within fingerprints.

Figure 3 presents the overall distribution of devices containing at least one identifier from the following list: *Personally Identifiable Information* (PII) including emails, emergency phone numbers and contact names, *Network identifiers*-MAC addresses-, and *Persistent Identifiers* including serial numbers, IMEI/MEIDs, IMSIs, and ICCIDs. Specifically, Figure 3a shows the distribution of devices containing at least one identifier by Android version while Figure 3b shows the distribution across different manufacturers.

In total, 25.96% of devices in our dataset contain at least one identifier among the considered types.

The majority of the detected identifiers are MAC addresses (18.45%), followed by serial numbers (5.89%) and Email addresses (4.07%).

Interestingly, we observe that the percentage of detected identifiers decreases with newer Android versions, indicating improved privacy measures and more restrictive access to sensitive APIs in recent Android releases, particularly following recent changes introduced by Android [9]. For instance, over 87.67% of Android 9 devices of our dataset expose at least one identifier, whereas this percentage drops significantly for Android 15 and 16. On the other hand, for manufacturer-specific distributions, Xiaomi and OnePlus stand out, with 54.89% and 47.92% of their devices exposing at least one identifier, respectively. In contrast, brands like Google show lower exposure percentages.

5.1 Personally Identifiable Information (PII)

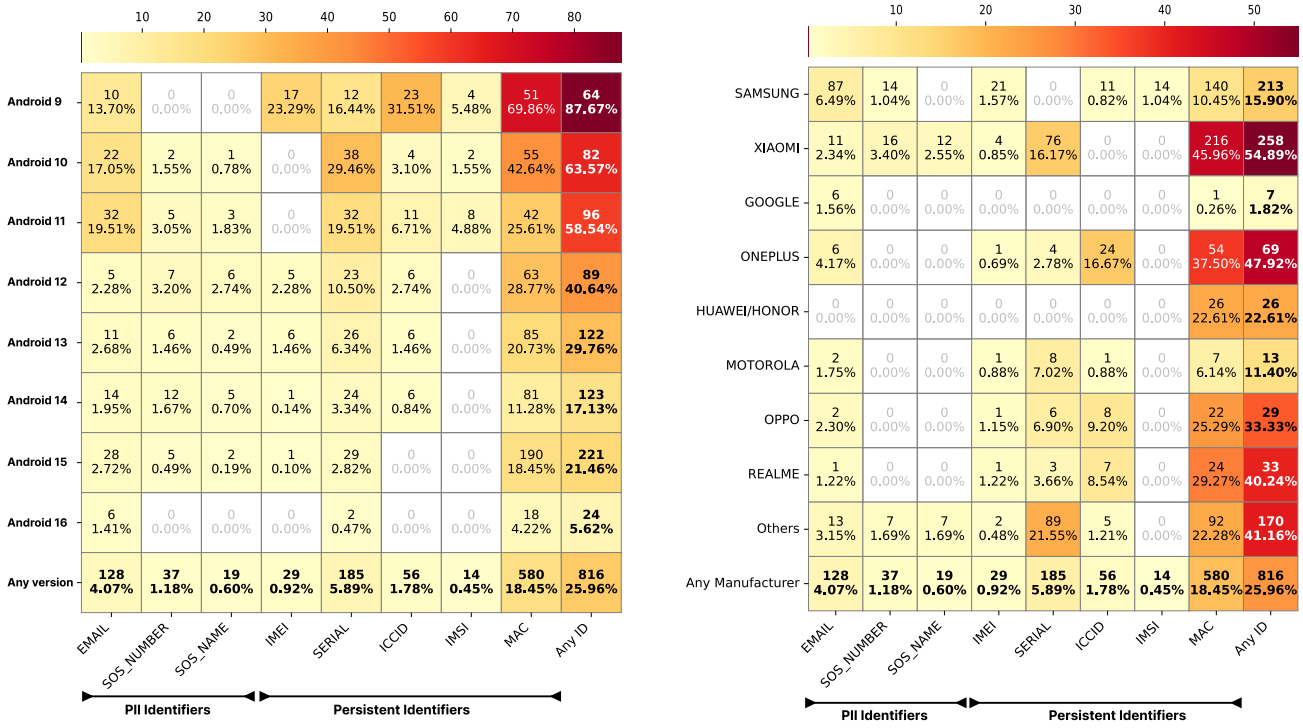
5.1.1 Email Addresses. We processed our fingerprints using regular expression filtering to identify attributes that could potentially expose user's email addresses. Our filter detected 17 access channels. A complete list of detected access channels is provided in Table 4 in Appendix B. In total, we identified 151 unique email addresses coming from 128 distinct devices. Notably, Samsung devices accounted for the majority of them, with 6.49% of all Samsung devices containing at least one email address.

While some devices leaked email addresses due to vendor-specific customizations, we were surprised to observe that some emails were accessible through `AccountManager.getAccounts`[6] API. This method is part of the official Android framework and is protected by the `GET_ACCOUNTS` permission, which is classified as `"Dangerous"` permission. The majority of devices leaking email addresses through this API were Samsung phones, where the email was linked to the Samsung package `com.osp.app.signin`. On devices from other manufacturers, the leaked email address was often associated with non-market apps (applications installed from downloaded APKs) such as the application `app.revanced`.

5.1.2 Emergency Contacts and fingerprint's nicknames. We ran regular expressions against our database in case our semi-exhaustive methodology collected personal data. One hit we had was regarding some phone numbers that were in clear text stored under a non-documented and custom *Settings.Secure*[7] preference, `key_miui_sos_emergency_contacts`, on 16 Xiaomi phones. After analyzing the API in more details, we found out that the numbers belonged to emergency contacts registered on the smartphones and another attribute named `key_miui_sos_emergency_contacts_names` is storing the names of these contacts. We also found emergency phone numbers on Samsung devices, 14 of them customized the default emergency numbers (default values are for example 112 for European countries, 911 for Americans, ...) under two custom *Settings.Secure* preferences, `emergency_[selected,custom]_number`, that are defined by the Samsung package `com.samsung.android.emergency`. In total, we have detected 57 different phone numbers and 34 different contact names.

Additionally, we found on 302 Xiaomi devices (64.25%) the nicknames belonging to the fingerprints used to unlock the phone, stored under *Settings.Secure* custom preference `settings_fingerpr`

[6] https://developer.android.com/reference/android/accounts/AccountManager#getAccounts()
[7] https://developer.android.com/reference/android/provider/Settings.Secure



(a) Device Identifiers Distribution By Android Versions

(b) Device Identifiers Distribution By Manufacturers

Figure 3: Device Types Identifiers Distribution by (a) Android Versions, (b) Manufacturers.

`int_id_prefix_`, which we observed exclusively on Xiaomi devices. While 79.13% retained their default names like "Fingerprint 1" or "Fingerprint 2" in their respective language, some users explicitly mentioned the person the fingerprint belongs to, while others indicated the exact finger associated with it, like "thumb left" or "index right". While these nicknames should not be accessible without any permission, the use of the non-default names rendered their users immediately identifiable in our database. In the end, vendor customization from Xiaomi leaks unique information on the user.

5.2 Persistent Identifiers

To detect persistent identifiers in our dataset, we focused on four key types of identifiers defined in the Android documentation [9]: device Serial Number IMEI/MEID, IMSI and ICCID. Additionally, we included MAC addresses, which uniquely identify hardware components, and wifi SSIDs, which can facilitate location tracking and user profiling.

5.2.1 MAC Addresses. We extracted a total of 2,743 potential MAC addresses from 580 different devices by applying regular expression filtering on our collected fingerprints. On average, each device contains 9.45 addresses, while the median value is 2.

In total, we identified 83 different access channels responsible for leaking these addresses. The majority of them are either **hidden APIs**, present in the AOSP source code but undocumented, or **custom APIs** introduced by manufacturers. A summary of some of these access channels is provided in Table 5 in Appendix B.

It is important to note that 80.56% of the extracted MAC addresses correspond to Bluetooth MAC addresses of connected devices, such as headphones. For instance, the hidden preference key `bluetooth_headset_priority_[MAC_addr]` of *Settings.Global*[8] stores the priority level of a paired Bluetooth headset along with its MAC address[9].

Notably, we also identified one documented API, `AudioManager.getCommunicationDevice().getAddress()`[10], which exposes the MAC address of the currently connected audio device used for communication. This API is not protected by any permission, making it a potential privacy concern. Moreover, we identified in our dataset two different devices sharing the same MAC address for the connected Bluetooth device, suggesting a possible cross-linking risk between devices and thus allowing inference of associations between users.

While we note that the collected addresses pose a privacy problem, the goal of this paper is not to understand the behavior of MAC address changes in depth and is left for future work.

5.2.2 Wifi SSID. A wifi SSID is the name of a wifi network. It helps devices identify wireless networks and connect to them. Before Android 10, it was possible to call the `WifiConfiguration.getConfi`

[8] https://developer.android.com/reference/android/provider/Settings.Global
[9] https://android.googlesource.com/platform/frameworks/base/+/refs/heads/main/core/java/android/provider/Settings.java
[10] https://developer.android.com/reference/android/media/AudioManager#getCommunicationDevice()

guredNetworks()[11] method to get the list of all wifi networks saved on a device. However, because the list was highly unique between users, this method ended up being deprecated. In our dataset, we identified two manufacturers which provide unrestricted access to specific wifi SSIDs:

Samsung devices have a feature called "Auto wifi" [37] which turns wifi on and off depending on the user's location. Accessing the `content://settings/global.sem_auto_wifi_added_removed_list` attribute provides the list of all added and removed wifi hotspots stored on the user's device. 403 devices (30.07% of Samsung devices) in our dataset have at least one SSID in their list and 387 of them (96.02%) have a unique list.

Vivo devices give access to a custom `WifiManager.getExtWifiScanResults` method which provides a list of the latest scanned Wifi hotspots without requiring any permission. This method is a copy of the documented `WifiManager.getScanResults`[12] which normally needs two permissions to function `ACCESS_WIFI_STATE` and `ACCESS_FINE_LOCATION`. This method was present in 65 out of 88 Vivo fingerprints and executable without permissions in 9.

In both cases, the methods seem to be helper functions for additional features of the phones but their unintended side effect is that they leak highly unique information about users and their location.

5.2.3 Documented Identifiers. In 2023, Meng et al. [29] reported on 51 access channels that leak *User-unresettable Identifier*s (UUIs) on Android. To identify persistent identifiers within our dataset, we assumed that these access channels are valid and can effectively retrieve persistent identifiers. Furthermore, to identify new access channels, we implemented the following **recursive process** that continues searching until no new access channels are discovered:

1) We scan all fingerprints to detect the presence of known access channels as reported by Meng et al. [29].

2) If an access channel is found, we extract its corresponding value within the current fingerprint and check for its occurrences in the same fingerprint. This step enables the discovery of potentially new access channels.

3) If new access channels are detected, they are added to the initial search list, and the process repeats from Step 1. The recursion continues until no additional access channels are found.

This extraction process revealed 362 potential persistent identifiers leaked across 266 different devices. For instance, we identified **13 of the 51 access channels** previously reported by Meng et al. and more importantly, we discovered **30 new channels potentially** leaking persistent identifiers. A summary of some newly discovered access channels is provided in Table 6 in Appendix B.

One notable example is the custom preference key `ddssim_iccid` found in *Settings.Global*, which leaks ICCID values from 23 OnePlus devices running Android versions 10 to 14.

Interestingly, we also identified a documented API, `TelecomManager.getUserSelectedOutgoingPhoneAccount().getId()` [13], which leaks the ICCID on devices running Android 9 (API level 28). Although this API is officially released and documented starting from API level 29, it existed in the AOSP source code earlier as a hidden API and likely unfinished feature under development. This suggests that some APIs under development or not yet officially released may accidentally expose sensitive data.

[11] https://developer.android.com/reference/android/net/wifi/WifiManager.html#getConfiguredNetworks()

[12] https://developer.android.com/reference/android/net/wifi/WifiManager#getScanResults()

[13] https://developer.android.com/reference/android/telecom/TelecomManager#getUserSelectedOutgoingPhoneAccount()

6 Attribute Analysis

In this section, we highlight key attributes from our dataset that provide insights into the fingerprintability of the Android ecosystem. As detailed in Section 3.3.2, we use entropy to evaluate the impact of an attribute on fingerprint uniqueness. We computed it for each attribute we collected and report on a selection of the most revealing ones in Table 2 along with some examples of their values. To provide a better understanding of their structure and content, Table 7 in Appendix C provides additional example values for these attributes.

To understand the different factors that impact device fingerprinting, we categorize fingerprint attributes into three distinct groups: Hardware-Related, OS-Related, and User-Related attributes, and analyze how variations in these attributes can influence the fingerprint. It should be also noted that to avoid considering multiple fingerprints from the same device, we use either the ANDROID_ID, which uniquely identifies a user within an application, or our generated ID, and take only the last collected fingerprint for each device.

6.1 User-Related Attributes

User-Related attributes highlight the impact of user personal settings and behaviors on the fingerprint uniqueness. We observed that these attributes tend to show high variability and thus contribute significantly to fingerprint uniqueness. Below are examples that show how some user customization can render a fingerprint unique with a single value:

"Device Name": It exhibited high diversity with 1815 unique values among 3143 devices. Notably, many of these unique values expose *Personally Identifiable Information* (PII) such as the name of the user. 899 devices used the format "[Name]'s [Phone Model]" which is the default behavior exhibited by Samsung devices during their initial setup phase. In other cases, users manually customized their device name, sometimes replacing it with something completely different. The ones that are not unique in our database come from phones that only indicate the device model, without enticing users to change the name during the first boot.

"Device Ringtone": It is possible to collect the current device ringtone through the Ringtone API. We found out that 56.53% of devices present a unique ringtone in our dataset. There are two main reasons behind this high percentage. First, there is a high diversity of ringtones between vendors on top of the default provided ones. If a user chooses a ringtone that is unique to a vendor, the probability that someone else is sharing the same ringtone is less likely. Second, it is possible on Android to use a custom audio file for the ringtone that will make the user immediately identifiable as they are unique and not shared between devices. We also observe the same behavior with notification and alarm sounds as 44.86% and 26.05% of them were unique, further enriching their fingerprint.

"Network Operator" and **"SIM Operator"**: These two attributes offer a moderate combined entropy of 7.81, but they can provide

Table 2: Representative Fingerprint Attributes By Category.

Attribute	Category	Type	Examples	Distinct Values	Unique Values	Entropy
System Up Since	User	Shell	`"2024-10-28 09:35:21","2025-11-12 12:38:37"`	3121	3104	11.59
Web View Default User Agent	OS & Hardware	SDK	`"Mozilla/5.0 (Linux; Android 13; PGT-AL00 Build/ HUAWEIPGT-AL00; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/101.0.4951.61 Mobile Safari/537.36"`	2815	2582	11.37
Available Displays	OS & Hardware	SDK	`"Display id 0: DisplayInfo"Built-in Screen", ..., real 1080x2400, largest app 2142x2142, smallest app 1080x1006}, DisplayMetricsdensity=2.625, width=1080, height=2142,..."`	2384	2063	10.68
Device Name	User & Hardware	Content Provider	`"Patrick's S24+", "Pixel 6a", "realme narzo 30 5G"`	2145	1822	10.56
List of System Services	OS	Shell	`["HqmManagerService","knoxcustom", "SEAMService", "SurfaceFlingerAIDL", ...]`	2029	1575	10.54
Root Dir Total Space (Bytes)	OS	SDK	`1010438144, 982638592`	1911	1413	10.45
IMEI/MEID Type Allocation Code	Hardware	SDK	`86326005, 35950275`	1978	1486	10.25
Device Ringtone	User & OS	Content Provider	`content://media/external/audio/media/1000101?title=Mission %20Impossible%20(Ti%C3%ABsto%20Remix%20Radiocanonical=1")`	2046	1777	10.06
Kernel Information	OS	Shell	`"Linux localhost 5.10.224-Kirisakura_Raviantah_2.5.1 #1 SMP PREEMPT Sat Oct 19 19:58:02 UTC 2024 aarch64 Toybox"`	1797	1348	10.05
Memory Usable By Apps (MB)	OS & Hardware	Shell	`"5601 x 3071", "7607 x 3071" (Total x Swap)`	1331	839	9.56
Boot Count	User	Content Provider	`1664`	602	255	8.21
Communication Device Name	User	SDK	`"LG HBS810", "WH-CH510", "JBL LIVE460NC"`	944	543	8.04
Network and SIM Operators	User	SDK	`("PROMOBILE SN", "Orange"), ("AT&T", "AT&T"), ("NL KPN", "Lebara")`	806	486	7.81
Next Alarm Time	User	Content Provider	`"Fri 16:27" , "ven. 06:53", "tors 06:00"`	1295	1094	5.85

strong indicators about where the user is living and if they are traveling in another country. For example, region-specific carriers like "SFR" in France or "Deutsche Telekom" in Germany are often only chosen for people residing in these countries. On the other hand, virtual providers like "Lycamobile" that provide cheap mobile plans for international travel can indicate the user is not a resident of the visited country. Finally, combined with other attributes, they help reduce the pool of potential device identities.

"System Up Since" and **"Boot Count"**: The first attribute indicates the device's last boot time, while the second indicates the number of boots. While it could be brittle to utilize one or the other to identify a device with a possibility of collisions, using both attributes in our dataset achieves a 99.78% rate of identification. Users with older devices are more prone to identification than more recent ones. For example, we found a device running below API level 30 having a number of reboots around 1664. Since API level 30 was released around 2020, this high number indicates the user is shutting down their phone every day, making it more at risk of being identified because few will have a number as high as theirs.

"Next Alarm Time": It is possible to collect when the next alarm will be triggered either through the *AlarmManager* API[14], or through the deprecated but still accessible entry `"NEXT_ALARM_FORMATTED"` of *Settings.System* [15]. While the next alarm may be a one-time one, a collection of several alarms over a week can reveal daily routines and user habits, further differentiating one device from another. The collected string is even in the primary language used by the user on their smartphones.

6.2 OS-Related Attributes

OS-related attributes reflect the software environment of the device, and highlight the impact of variability introduced by Android versions, system configurations, and vendor customization. As shown in Table 2, these attributes show high entropy, making them highly discriminative. Below are some concrete examples:

"WebView Default User Agent": This attribute is one taken from the field of browser fingerprinting where we capture the User Agent given by WebView, which is the in-app browser for Android applications. It combines several information about the software such as the Android version and the build details, as well as hardware-related details like the device model. It is highly discriminative, with 2582 unique values among 3143 devices.

"Available Displays": It gives information about both hardware (e.g., screen resolution) and software (e.g., app window dimensions),

[14]https://developer.android.com/reference/android/app/AlarmManager#getNextAlarmClock()

[15]https://developer.android.com/reference/android/provider/Settings.System#NEXT_ALARM_FORMATTED

leading to a high entropy (10.68). On the same hardware, diversity can appear from different OSes providing different UI dimensions.
"List of System services": It provides all the services installed on the device and we noticed a large diversity of them across all vendors. We identified in our dataset 1397 different services: 75 (5.36%) appeared on all devices and 828 (59.26%) only on devices from a single vendor. 573 (41.01%) services are present on 10 devices or less. While some reflect the capabilities of a device like `knoxguard_service` for devices equipped with the Samsung Knox security solution or `asus_game_mode` for Asus phones providing a special game mode, other services are present because of specific user actions. For example, the `rcs` service implies that the user has activated Rich Communication Services on their phone, which is an evolution of the traditional SMS. Another is `music_recognition` which is a service running in the background to identify what music track is playing near the user. All these services unfortunately enhance how unique a fingerprint can be.
"Kernel Information": It reflects direct information about the OS build, such as the kernel version and build date, which can reveal whether the device runs a stock ROM or a custom ROM.

6.3 Hardware-Related Attributes

Hardware-related attributes are more stable as they are linked directly to the physical components of the device. One example is the **IMEI/MEID Type Allocation Code (TAC)**, which represents the first eight digits of the IMEI/MEID that identifies the device model and manufacturer. As shown in Table 2, this attribute shows a relatively high number of distinct values with 1978 distinct values, making it a strong indicator of device type. While TAC does not uniquely identify a device on its own, it significantly contributes to fingerprinting when combined with other attributes.

7 Fingerprintability of the Android ecosystem

As mentioned, some specific attributes render a device instantly recognizable, through, for example, a custom user setting, a vendor customization, or a unique identifier. In this section, we adopt a more global approach by considering how fingerprint-based tracking could be performed in the wild. If a tracker were to design an Android device fingerprinting algorithm to identify as many devices as possible, what attributes would they use? How large would the fingerprints be? The following analysis focuses on two main aspects to address this question: the **uniqueness** of device fingerprints and their **stability** over time.

To assess **fingerprint uniqueness**, we use the same entropy-based metric previously applied to individual attributes. In addition, we compute the normalized entropy score for every attribute, which allows us to evaluate the diversity of attribute distributions on a uniform scale (from 0 to 1). This helps identify which attributes contribute the most to device identification.

We begin by preprocessing our dataset. First, we eliminate **unchanged** attributes (constant attributes that do not provide any information to distinguish devices) as well as **globally unstable** attributes across all devices, that is to say attributes that change for all devices and are therefore unsuitable for fingerprinting. This cleaning step results in a set of 55,665 attributes.

Second, we retain only one representative attribute among groups of attributes that share identical values and distributions across the dataset, which further reduces the attribute set to 12,060 attributes. Finally, we keep only attributes whose individual entropy is greater than "0.5", ensuring a sufficient level of variability to be useful for fingerprinting but not enough to identify devices on its own. This leads to a final set of 220 attributes.
Then, we evaluate how efficiently devices can be uniquely identified as more attributes are combined from the set of 220 high-entropy attributes.
Figure 4 illustrates how uniqueness evolves as the number of attributes used in the fingerprinting algorithm increases. The x-axis represents the number of attributes, while the y-axis shows the highest percentage of unique fingerprints achieved with the optimal combination for a given number of attributes.

Interestingly, using only the high-entropy "System up since" attribute, we achieve a uniqueness score of 98.75%. Moreover, we observe that a combination of just **five** attributes suffices to reach 100% uniqueness score. While these results demonstrate a very high identification capability, many of the most discriminating attributes, such as the "System up since", are not stable over time. Such attributes may allow short-term identification, but fail to support long-term tracking at scale. To address this, we analyze fingerprint stability, which measures how consistently a fingerprint remains the same across multiple scans from the same device.

Fingerprint stability analysis requires multiple fingerprints per device. In our dataset, 440 devices scanned their fingerprints at least twice, resulting in a total of 1301 fingerprints. The maximum time interval between two scans of the same device is 345 days (approximately one year), enabling us to track long-term attributes changes.

We start by evaluating the stability of each attribute individually using its **inter-device change count** $C_{\text{inter}}(A)$, which measures how often an attribute exhibits at least one value change across all devices. Based on this metric, we construct subsets of attributes that remain stable for at least 90%, 95%, 99%, and 100% of the devices. And we recompute device uniqueness using these stability-constrained subsets and compare the results with those obtained using only high-entropy attributes (without imposing additional stability constraints beyond the initial preprocessing).

This comparison is illustrated in Figure 4. Combining **five** attributes suffices to reach 100% uniqueness across devices. When restricting the analysis to attributes that remain stable on at least 99% of the devices, **nine** attributes are required to achieve near full uniqueness (99.68%). Among these, two stand out due to their combination of high entropy and high stability: *Device Name* and *Network Operator*. Although these attributes may be influenced by user behavior, they remain consistent for 99% of the devices. In contrast, when considering only attributes that are stable on all devices, uniqueness increases more gradually. With a single attribute, uniqueness is close to 22.91%, and we reach 99.62% with a combination of **28** fully stable attributes.

Overall, while high-entropy attributes enable near-perfect short-term identification with very few attributes, enforcing strong stability constraints still allows nearly perfect identification using a small, carefully selected subset of stable attributes.

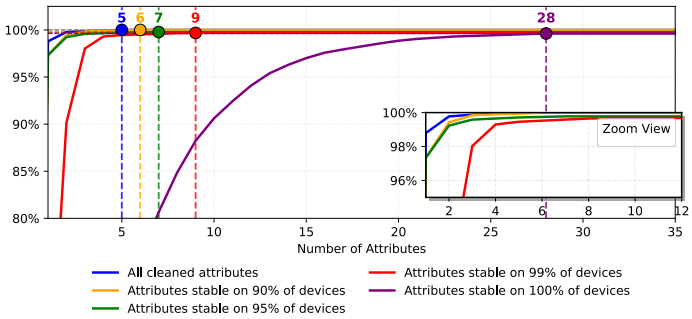


Figure 4: Evolution of the rate of fingerprint uniqueness based on the number of selected attributes. For each set of attributes, the combination that led to the highest uniqueness rate was selected.
For example, when considering attributes stable on 99% of the dataset (red curve), a combination of nine attributes achieves a maximum uniqueness of 99.68%.

8 Discussion

8.1 Main takeaways

Device fingerprinting is a threat to Android users and goes beyond what is possible through a web browser. When we started our experiments, we did not know what we would find out. One point of comparison we had was the field of browser fingerprinting which presented varying results over the years. One study by Laperdrix et al. [28] showed that 81% of 13,105 mobile browser fingerprints were unique with 17 attributes. Gómez-Boix et al. [19] had this number down to 18.5% from 251,166 mobile browser fingerprints. Even though the scale of our data collection is smaller than the ones cited before, we actually did not expect to be able to uniquely identify every single device of our database without a single permission with device fingerprints. Browsers have a very strict set of standards that all browser vendors follow, which results in controlled and limited browser fingerprints. For Android, the reality is much different. The open nature of the OS coupled with the richness of the features provided by modern smartphones mean that a lot more information can transpire inside a device fingerprint and make the user more easily identifiable. We hope that our study brings the necessary attention to device fingerprinting so that progress can be made to limit the leak of information and improve privacy for all Android users.
The more the user customizes and utilizes their phone, the more identifiable they are. Customizing the device name, the ringtone or the text highlight color, having applications that spawn background services like music recognition, pairing one or two Bluetooth devices, installing a custom Android OS. These examples are simple and reflect normal phone usage but the fingerprints we collect can capture these changes and they render users a lot more identifiable than they should be. This is very similar to what can be found in the field of browser fingerprinting. Users on Linux having a non-Chrome browser with very specific extensions to safeguard their privacy are a lot more recognizable than users on Windows using an unmodified Chrome version. In the case of Android here, users who deviate from the most popular phones and the default configuration start to get more and more recognizable.
Too many attributes are not protected by default. While getting access to the user's ringtone or the number of reboots of the device can appear as benign, these actions take a whole different meaning from a device fingerprinting perspective. A lot of attributes that we can collect are direct and sometimes indirect results of a user's action and they should be protected behind a permission. For example, the Runtime API that we use to launch shell commands should be put behind a permission. The Settings content provider that we rely on to access several system settings is read-only without a permission but we argue that it should even be protected with the type of information it can provide.
Vendor customization greatly enhances device fingerprinting. The data we obtained from APIs that were added by vendor customization can have harmful effects on user's privacy. If it is not additional attributes that provide additional ways for a tracker to differentiate one device from another, it can provide personally identifiable information like an email address or an IMEI number as detailed in Section 5. With our study, we echo the findings of other works like [29] that found identifiers in attributes provided by the added vendor layer.

8.2 Mitigating device fingerprinting

One way that users can limit tracking on their Android devices is to delete their Advertising ID [14]. In light of our results, this action now feels limited as a proper device fingerprinting algorithm could recognize the device without the need for such an ID. We suggest below three directions that could mitigate device fingerprinting on Android.
Making changes to high entropy attributes. One immediate solution to the problems identified in this study is to put the attributes revealing the most information behind a permission. To go even further, one additional step would be to reduce the granularity of the presented information to its bare minimum for its intended functionality. For example, kernel information could be limited to the version number without mentioning its exact build time.
Controlling vendor customization. Because of the problems reported on vendor customization, there should be mandatory controls on Android to make sure that sensitive and protected information does not end up being copied and available without any restrictions.
Identifying the privacy practices of an app dependency. In 2023, Apple introduced Privacy Manifest files [12] for iOS applications. In it, developers need to describe the data their app and their third-party SDKs collect along with the reasons why specific APIs are accessed. Without this file, an app cannot be submitted to the App Store. While this requirement does not fix aggressive data collection, it does improve transparency for users but also for developers who may be unaware of some of the data collected by a third-party SDK. A similar system for Android could provide some improvements if developers are unwilling to include dependencies that access APIs that are completely unrelated to the core purpose of their own application.

8.3 Limitations and future work

The main limitation of our study is the small scale of our dataset. While we found that device fingerprinting is very effective on the pool of fingerprints we collected, the question remains if all or only some of our findings would still hold at a much larger scale. Hardware and device-related attributes would potentially present more collisions as more users would have the same phone with the same Android version. Yet, user-related ones would still provide information that could potentially uniquely identify someone. Without a much bigger number of participants, it is hard to predict how these changes would truly affect the fingerprintability of the Android ecosystem.

For future work, one direction is to improve the SDK attribute extraction process. In particular, expanding the list of explored classes by incorporating vendor-specific classes could uncover additional fingerprinting attributes. Furthermore, improving the input parameters generation process when instantiating classes or invoking methods, by using fuzzing techniques for example, could provide more realistic arguments. Beyond improving our semi-exhaustive fingerprinting method, this work can also be extended by collecting information that is protected by specific permissions to understand how much threat they can pose to privacy. Notably, all the normal permissions that do not require explicit consent from the user as they are not deemed dangerous would be a great start. A final research direction would be to investigate in depth how device fingerprints on Android evolve over time. Since each attribute that we collect can change in their own way, knowing how they evolve would help us better protect users against trackers whose aim would be to link a newly received fingerprint with a profile of a known user. However, this direction requires a new experiment enrolling users for recurring visits so that we can collect meaningful data over a set period of time.

9 Availability

We have made the EXADPrinter framework source code, along with data analysis scripts for cleaning and preprocessing the dataset at https://github.com/AmIUniqueTools. The AmIUnique application can be found on the Google Play Store at https://play.google.com/store/apps/details?id=com.amiunique.amiuniqueapp.

10 Conclusion

In this work, we present EXADPrinter, a framework for Android devices performing device fingerprinting without a single permission. By exploring a wealth of APIs and commands available for an application, EXADPrinter builds fingerprints composed of more than 200,000 attributes. Analyzing 4,004 fingerprints collected from a mobile application reveals that device fingerprinting is extremely effective on Android with only a few attributes needed to identify all the devices of our dataset. As the collective arms race against online tracking continues, we hope that our tool and findings will help improve the privacy of the Android platform.

Acknowledgments

This work has been financially supported by the Agence Nationale de la Recherche through the ANR-21-CE39-0019 FACADES, the ANR-22-PECY-0002 IPoP and the ANR-22-PTCC-0001 SWHSec projects. We also thank all the users of the AmIUnique application that made this study possible. The authors used AI-based tool ChatGPT, to revise the text, correct any typos, grammatical errors, and awkward phrasing.

References

[1] Ali Alkinoon, Trung Cuong Dang, Ahod Alghuried, Abdulaziz Alghamdi, Soohyeon Choi, Manar Mohaisen, An Wang, Saeed Salem, and David Mohaisen. 2025. A Comprehensive Analysis of Evolving Permission Usage in Android Apps: Trends, Threats, and Ecosystem Insights. *Journal of Cybersecurity and Privacy* 5, 3 (2025), 58.

[2] Stephen Almond. 2024. Our response to Google's policy change on fingerprinting | ICO. Retrieved November 01, 2025 from https://ico.org.uk/about-the-ico/media-centre/news-and-blogs/2024/12/our-response-to-google-s-policy-change-on-fingerprinting/

[3] Android. 2025. Android API levels cumulative usage. Retrieved November 01, 2025 from https://apilevels.com/

[4] Android. 2025. Android Content Providers. Retrieved November 01, 2025 from https://developer.android.com/guide/topics/providers/content-providers

[5] Android. 2025. Android Debug Bridge (ADB). Retrieved November 01, 2025 from https://developer.android.com/tools/adb

[6] Android. 2025. Android Developer Reference. Retrieved November 01, 2025 from https://developer.android.com/reference

[7] Android. 2025. Android Identifier ANDROID_ID. Retrieved November 01, 2025 from https://developer.android.com/reference/android/provider/Settings.Secure#ANDROID_ID

[8] Android. 2025. *Android Open Source Project (AOSP).* Retrieved November 01, 2025 from https://source.android.com/docs/setup/about

[9] Android. 2025. Device Identifiers in Android. Retrieved November 01, 2025 from https://source.android.com/docs/core/connect/device-identifiers

[10] AppBrain. 2025. *Number of available applications in the Google Play Store as of November 2025.* Retrieved November 01, 2025 from https://www.appbrain.com/stats/number-of-android-apps

[11] Apple. 2025. Apple Developer Program License Agreement. Retrieved November 01, 2025 from https://developer.apple.com/support/terms/apple-developer-program-license-agreement/

[12] Apple. 2025. Privacy manifest files - Describe the data your app or third-party SDK collects and the reasons required APIs it uses. Retrieved November 01, 2025 from https://developer.apple.com/documentation/bundleresources/privacy-manifest-files

[13] BrowserStack. 2025. BrowserStack Testing platform. Retrieved November 01, 2025 from https://www.browserstack.com/

[14] Bennett Cyphers. 2022. How to Disable Ad ID Tracking on iOS and Android, and Why You Should Do It Now | EFF. Retrieved November 01, 2025 from https://www.eff.org/deeplinks/2022/05/how-disable-ad-id-tracking-ios-and-android-and-why-you-should-do-it-now

[15] Zikan Dong, Tianming Liu, Jiapeng Deng, Li Li, Minghui Yang, Meng Wang, Guosheng Xu, and Guoai Xu. 2024. Exploring covert third-party identifiers through external storage in the android new era. In *33rd USENIX Security Symposium (USENIX Security 24).* 4535–4552.

[16] Peter Eckersley. 2010. How unique is your web browser?. In *Privacy Enhancing Technologies: 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings 10.* Springer, 1–18.

[17] European Parliament and Council of the European Union. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation). , 88 pages. Retrieved November 01, 2025 from https://eur-lex.europa.eu/eli/reg/2016/679/oj

[18] Christof Ferreira Torres and Hugo Jonker. 2018. Investigating fingerprinters and fingerprinting-alike behaviour of android applications. In *European Symposium on Research in Computer Security.* Springer, 60–80.

[19] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the crowd: an analysis of the effectiveness of browser fingerprinting at large scale. In *Proceedings of the 2018 world wide web conference.* 309–318.

[20] Google. 2024. Updating our platform policies to reflect innovations in the ads ecosystem. Retrieved November 01, 2025 from https://support.google.com/marketingplatform/answer/15732590?hl=en

[21] Google. 2025. Advertising ID AD_ID. Retrieved November 01, 2025 from https://support.google.com/googleplay/android-developer/answer/6048248

[22] Kris Heid and Jens Heider. 2024. Haven't we met before?-Detecting Device Fingerprinting Activity on Android Apps. In *Proceedings of the 2024 European Interdisciplinary Cybersecurity Conference.* 11–18.

[23] Thomas Hupperich, Davide Maiorca, Marc Kührer, Thorsten Holz, and Giorgio Giacinto. 2015. On the robustness of mobile device fingerprinting: Can mobile

users escape modern web-tracking mechanisms?. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 191–200.
[24] Brian Kondracki, Babak Amin Azad, Najmeh Miramirkhani, and Nick Nikiforakis. 2022. The droid is in the details: Environment-aware evasion of android sandboxes. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*.
[25] Landley.net. 2025. Toybox. Retrieved November 01, 2025 from https://landley.net/toybox/help.html
[26] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. 2020. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)* 14, 2 (2020), 1–33.
[27] Pierre Laperdrix, Naif Mehanna, Antonin Durey, and Walter Rudametkin. 2022. The price to play: A privacy analysis of free and paid games in the android ecosystem. In *Proceedings of the ACM Web Conference 2022*. 3440–3449.
[28] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 878–894.
[29] Mark Huasong Meng, Qing Zhang, Guangshuai Xia, Yuwei Zheng, Yanjun Zhang, Guangdong Bai, Zhi Liu, Sin G Teo, and Jin Song Dong. 2023. Post-GDPR Threat Hunting on Android Phones: Dissecting OS-level Safeguards of User-unresettable Identifiers.. In *NDSS*.
[30] Oracle. 2025. Java Reflection. Retrieved November 01, 2025 from https://www.oracle.com/technical-resources/articles/java/javareflection.html
[31] Gerald Palfinger and Bernd Prünster. 2020. Androprint: analysing the fingerprintability of the android api. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 1–10.
[32] Yuan Yang Patrick McGee. 2025. China's tech giants test way around Apple's new privacy rules | Financial Times. Retrieved November 01, 2025 from https://www.ft.com/content/520ccdae-202f-45f9-a516-5cbe08361c34
[33] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, Phillipa Gill, et al. 2018. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In *The 25th annual network and distributed system security symposium (NDSS 2018)*.
[34] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 2019. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX security symposium (USENIX security 19)*. 603–620.
[35] Jingjing Ren, Martina Lindorfer, Daniel J Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. 2018. A longitudinal study of pii leaks across android app versions. In *Network and Distributed System Security Symposium (NDSS)*, Vol. 10.
[36] World Population Review. 2025. Mobile Phone Brands by Country 2025. Retrieved November 01, 2025 from https://worldpopulationreview.com/country-rankings/mobile-phone-brands-by-country
[37] Samsung. 2025. Intelligent Wi-Fi – Samsung Knox Documentation. Retrieved November 01, 2025 from https://docs.samsungknox.com/admin/knox-platform-for-enterprise/kbas/kba-360034073174/
[38] Justin Schuh. 2019. Building a more private web | Google. Retrieved November 01, 2025 from https://blog.google/products/chrome/building-a-more-private-web/
[39] Michael A Specter, Mihai Christodorescu, Abbie Farr, Bo Ma, Robin Lassonde, Xiaoyang Xu, Xiang Pan, Fengguo Wei, Saswat Anand, and Dave Kleidermacher. 2025. Fingerprinting SDKs for Mobile Apps and Where to Find Them: Understanding the Market for Device Fingerprinting. *arXiv preprint arXiv:2506.22639* (2025).
[40] StatCounter Global Stats. 2025. *Mobile Operating System Market Share Worldwide*. Retrieved November 01, 2025 from https://gs.statcounter.com/os-market-share/mobile/worldwide
[41] Imdad Ullah, Roksana Boreli, and Salil S Kanhere. 2023. Privacy in targeted advertising on mobile devices: a survey. *International Journal of Information Security* 22, 3 (2023), 647–678.
[42] Antoine Vastel, Pierre Laperdrix, Walter Rudametkin, and Romain Rouvoy. 2018. Fp-stalker: Tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 728–741.
[43] Wenjia Wu, Jianan Wu, Yanhao Wang, Zhen Ling, and Ming Yang. 2016. Efficient fingerprinting-based android device identification with zero-permission identifiers. *IEEE AcceappleManifestss* 4 (2016), 8073–8083.

A Additional Details on Methodology

A.1 Domain Exploration Key findings

Using 28 Android devices available on the testing platform BrowserStack[16], we investigate the presence and structure of scrapped classes using Java Reflection. Specifically, for each extracted class, we attempted to load it, by name, using the `Class.forName` method. If the class is not found, a `ClassNotFoundException` is thrown; otherwise, we proceeded to enumerate its members. For that, we used `Class.getFields()` and `Class.getMethods()`, which return the public fields and methods of the class, respectively. Here are some key observations when comparing the list of officially documented APIs and those found by Java Reflection:

- Some classes released and documented for API level X were also discovered on devices running earlier API levels, despite being undocumented. This discrepancy appears to stem from classes that were still under development or undergoing testing, and were only formally released in the subsequent version. To determine their earliest appearance in the Android Open Source Project (AOSP), we used the Android Code Search GUI.[17]
- A significant number of undocumented APIs (fields and methods) can be listed using Java Reflection. These APIs are internal and often marked with the `@hidden` annotation.
- Some manufacturers have introduced their own custom APIs, identifiable by their naming conventions, which include special keywords such as `SEM` for Samsung Electro-Mechanics and `MIUI` for Xiaomi's custom ROM. listing 1 below shows examples of some manufacturer custom APIs.

Listing 1: Manufacturer custom APIs examples.

```
# Samsung:
DigitalClock.SEM_DICTIONARY_ID
UserManager.semHasBaseUserRestriction
# OnePlus:
Build.DEBUG_ONEPLUS
Intent.getOplusUserId
# Xiaomi:
AppOpsManager.MIUI_OP_END
CellSignalStrengthGsm.getMiuiLevel
```

A.2 List of Shell Commands

Table 3 presents the set of selected shell commands used in our fingerprinting framework. Note that not all commands executed successfully, some of them returned errors, such as permission denied, depending on the device and system restrictions.

[16] https://www.browserstack.com/
[17] https://cs.android.com/

Table 3: List of Shell Commands and Descriptions.

Command	Description
`acpi -V`	ACPI battery information
`arp -a`	ARP cache
`cat /etc/group`	Groups
`cat /etc/passwd`	User accounts
`cat /proc/cpuinfo`	CPU information
`cat /proc/meminfo`	Memory information
`cat /var/log/auth.log`	Authentication logs
`df -h`	Disk space usage
`dmesg -T | head -n 1000`	First lines of kernel logs
`dmesg -T | tail -n 1000`	Last lines of kernel logs
`dpkg -l`	Installed packages (Debian-based systems)
`dumpsys`	Android system service diagnostics
`free -m`	Memory usage
`getconf -a`	System configuration variables
`getprop`	Android system properties
`hostname`	Show hostname
`hwclock`	Hardware clock time
`ifconfig`	Network interfaces
`ls /`	Root directory structure
`ls /system/fonts`	Installed system fonts
`ls /system/media/audio/ringtones`	Installed ringtones
`lsb_release -a`	Linux distribution information
`lsblk`	Block devices (storage partitions)
`lscpu`	CPU architecture and details
`lshw`	Detailed hardware information
`lsmod`	Loaded kernel modules
`lspci`	PCI devices
`lsusb`	USB devices
`netstat`	Network statistics
`nproc`	Number of processors
`pm list packages`	All installed Android packages
`pm list permissions -g`	Permissions grouped by category
`pm list features`	Supported hardware/software features
`pm list staged-sessions`	Pending or in-progress staged sessions
`ps aux`	Running processes
`route`	Routing table
`sysctl -a`	Kernel parameters
`stty -a`	Terminal settings
`tty`	Current terminal device
`uname -a`	Kernel and OS information
`uptime -s`	System start time

A.3 SDK Attributes Extraction Process

Listing 2 shows the list of method name prefixes we selected. Methods are invoked only if their names begin with those prefixes. The list includes also manufacturer-specific prefixes such as *"semget"* and *"vivoget"*, which were discovered during our domain exploration step when inspecting the SDK structure of various Android devices available through the BrowserStack testing platform.

Listing 2: Selected method name prefixes.

```
"get", "is", "has", "can", "should", "must", "contain",
"count", "to", "as", "fetch", "my","query", "find", "size",
"length", "id", "state", "status", "flag", "index", "name",
"are", "level", "describe", "title", "type", "retrieve",
"semget", "vivoget", "oppoGet"
```

Figure 5 summarizes the process step by step of gathering **SDK attributes** values using Java Reflection.

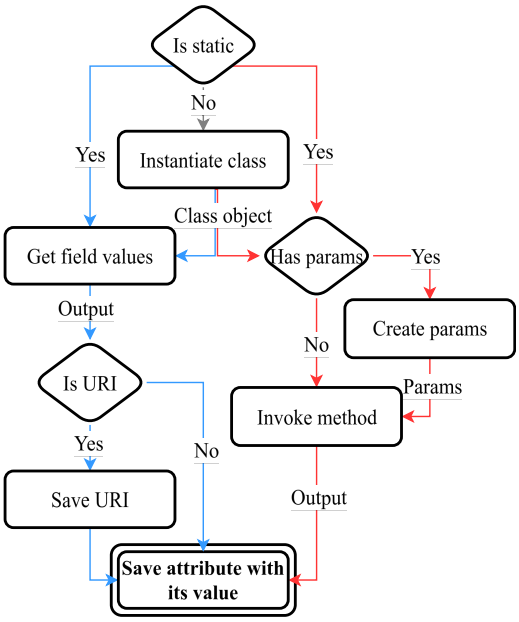


Figure 5: Process of getting SDK attributes.

A.4 Fingerprint Structure

Listing 3 shows a snippet of a fingerprint where we see the results of the three extraction techniques we have used for collecting attributes. The key *"uptime"* displays the output of the shell command `uptime -a`, while *"android.os.Build.BRAND"* and *"android.app.AlarmManager.getNextAlarmClock"* keys represent values obtained via Java Reflection by accessing fields and invoking methods. Finally, the key *"content://settings/system"* shows the result of dumping values from the **Settings.System** content provider.

Listing 3: Example of a fingerprint object.

```
{
   "uptime": "2025-02-11 17:03:48",
   "android.os.Build.BRAND": "samsung",
   "android.app.AlarmManager.getNextAlarmClock" :{
      "fields": [],
      "methods": [{ "getTriggerTime": "1728153120827" }]
   },
   "content://settings/system": [{
      "_id": "31809",
      "name": "device_name",
      "value": "XXX's S24",
   }]
}
```

B Discovered Access Channels

Table 4 summarizes the the discovered access channels leaking user email addresses, while Table 5 lists the top access channels leaking MAC addresses of connected devices. Both tables include the *API origin*, which indicates whether the access channel corresponds to a documented API, a hidden API (present in the AOSP source code but not officially documented), or a custom API introduced by device manufacturers or applications. In contrast, Table 6 presents newly discovered access channels that expose SIM card identifiers.

C Attributes Examples

Table 7 provides example values for a selection of fingerprinting attributes. For each attribute, we present examples from at least two different devices to highlight the observed diversity. We also give detailed information about each device, including the **manufacturer**, **model**, **Android version**, and the corresponding **OS** running on it.

Table 4: Summary of Attributes Leaking Emails Addresses by Manufacturers.

Attribute	API Origin	Affected Manufacturers (Total leaked emails)	Total Leaked Emails
AccountManager.getAccounts	Documented API	SAMSUNG (70), XIAOMI (10), GOOGLE (6), ONEPLUS (6), MOTOROLA (2), REALME (1),OPPO (1), Others (6)	114
Settings.System.samsungaccount	Custom API	SAMSUNG (22)	22
Settings.System.contact_default_account	Custom API	SAMSUNG (13)	13
Settings.Global.contact_default_account	Custom API	SAMSUNG (8)	8
Settings.System.sync_disabled_accounts	Custom API	SAMSUNG (5)	5
Settings.System.AccountListForCloud	Custom API	NUBIA (2)	3
Settings.Secure.smart_tethering_sa_account_id	Custom API	SAMSUNG (3)	3
Settings.Secure.sem_auto_wifi_added_removed_list	Custom API	SAMSUNG (2)	2
Settings.Secure.filter.accountSet	Custom API	VSMART (2)	2
Settings.Global.whitelist_mode_group_name	Custom API	VIVO (2)	2
Settings.System.contact_setting_list_filter	Custom API	SAMSUNG (2)	2
Settings.Secure.key_all_accounts	Custom API	VSMART (2)	2
Settings.Secure.miui_15_default_lockscreen_info	Custom API	XIAOMI (1)	1
Settings.Global.zen_mode_messages_group_name	Custom API	VIVO (1)	1
Settings.System.fp_ql_item_key_item_position_key_4	Custom API	OPPO (1)	1

Table 5: Example of Attributes Leaking MAC Addresses.

Attribute (Access Channel)	API Origin	Affected Manufacturers (Affected Devices Count)	Total Affected Devices Count
Settings.Global.bluetooth_a2dp_sink_priority_[MAC address]	Hidden API	SAMSUNG (43), XIAOMI (9) , HUAWEI/HONOR (23), ONEPLUS (9), OPPO (7), Others (14)	105
Settings.Global.bluetooth_headset_priority_[MAC address]	Hidden API	SAMSUNG (42), XIAOMI (9), HUAWEI/HONOR (22), ONEPLUS (9), OPPO (7), Others (14)	103
Settings.Global.bluetooth_input_device_priority_[MAC address]	Hidden API	SAMSUNG (37), XIAOMI (9), HUAWEI/HONOR (19), ONEPLUS (8), OPPO (7), Others (13),	93
Settings.Global.LEAUDIO_SECOND_KEY[MAC address]	Custom API	XIAOMI (67)	67
AudioManager.getCommunicationDevice().getAddress()	Documented API	SAMSUNG (25), XIAOMI (13), HUAWEI/HONOR (3), ONEPLUS (7), REALME (5), Others (10)	63
AudioManager.getCommunicationDevice().semGetAddress()	Custom API	samsung (7)	7

Table 6: Newly Detected Attributes Potentially Leaking Persistent Identifiers.

Identifier Type	Attribute (Access Channel)	Affected Manufacturers (Affected Devices Count)
ICCID	Settings.Global.ddssim_iccid[ICCID]	ONEPLUS (23)
	Settings.Global.oppo_comm_simsettings_daily_alert_snooze_	REALME (4), OPPO (1)
	TelecomManager.getUserSelectedOutgoingPhoneAccount().getId()	SAMSUNG (9), OPPO (3), Others (6)
	getprop.persist.vendor.radio.data.iccid	OPPO (5), MOTOROLA (1), Others (3)
	getprop.persist.vendor.radio.ls1icid	OPPO (4), REALME (1)
	getprop.persist.vendor.radio.ls2icid	OPPO (5), REALME (1)
	Settings.Global.data_limit_notification_bool[ICCID]	OPPO (2), REALME (2)
	Settings.Global.data_warning_notification_bool[ICCID]	OPPO (1), REALME (2)
IMSI	Settings.System.dsa_sim1_value	SAMSUNG (4)
	Settings.System.dsa_sim2_value	SAMSUNG (5)
Serial Number	getprop.persist.radio.factory_sn	XIAOMI (11)
	getprop.ro.boot.psn	MOTOROLA (3), LENOVO (3)

Table 7: Fingerprint Attributes Examples.

Attribute	Examples (Manufacturer, Model, API level)
System Up Since	`"2024-10-08 09:06", "2025-01-30 07:03"`
Web View Default User Agent	**(ITEL, itel P661N, 33):** `"Mozilla/5.0 (Linux; Android 13; itel P661N Build/TP1A.220624.014; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/129.0.6668.82 Mobile Safari/537.36"` **(Realme, RMX3242, 33):** `"Mozilla/5.0 (Linux; Android 13; RMX3242 Build/TP1A.220905.001; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/131.0.6778.260 Mobile Safari/537.36"`
IMEI/MEID Type Allocation Code	**(Realme, RMX3242, 33):** `86326005`, **(ITEL, itel P661N, 33):** `35640763`, **(Google, Pixel 6a, 34):** `35950275` **(Samsung, SM-A155F, 34):** `35273691`
Device Name	`"Patrick's S24+"`, `"S23 de Rafael"`, `"Pixel 6a"`, `"realme narzo 30 5G"` `"realme GT NEO 3T Dragon Ball Z"`
Available Displays	**(Google, Pixel 6a, 34):** `"Display id 0: DisplayInfo{\"Built-in Screen\", ... real 1080x2400, largest app 2142x2142, smallest app 1080x1006, ...}, DisplayMetrics{density=2.625, width=1080, height=2142,...}, isValid=true"` **(Google, Pixel 6a, 34) Lineage OS:** `"Display id 0: DisplayInfo{\"Built-in Screen\", ... real 1080x2400, largest app 2268x2205, smallest app 1080x943, ...}, DisplayMetrics{density=2.625, width=1080, height=2205,...}, isValid=true"` **(Samsung, SM-A155F, 34):** `"Display id 0: DisplayInfo{\"Yerleşik Ekran\", ... real 1080 x 2340, largest app 2125 x 2125, smallest app 1080 x 1012, ...}, DisplayMetrics{density=2.8125, width=1080, height=2125, ...}, isValid=true"`
List of System Services	**(Samsung, SM-S926U, 34):** `["HqmManagerService","SurfaceFlingerAIDL","SEAMService", "knoxcustom","semclipboard","semprivilege", ...]` **(Realme, RMX3242, 33):** `["MMListService","SurfaceFlingerAIDL","OPLUSExService", "oplusdevicepolicy", "opluscustomize", ...]` **(Xiaomi, 2203129G, 31):** `["MiuiBackup","MiuiCarService","MiuiInit","MiuiWifiService", "miui_step_counter_service", "miuiboosterservice",, ...]`
Root Directory Total Space (Bytes)	**(Google, Pixel 6a, 35):** `1010438144` **(Google, Pixel 6a, 34):** `982638592` **(Lineage OS, Pixel 6a, 34):** `1125883904`
Kernel Information	**(Google, Pixel 6, 35):** `"Linux localhost 5.10.214-android13-4-00006-geda2373a76b0 -ab12225542 #1 SMP PREEMPT Tue Aug 13 18:16:59 UTC 2024 aarch64 Toybox"` **(Google, Pixel 6, 35) Kirisakura OS:** `"Linux localhost 5.10.224-Kirisakura_Raviantah_2.5.1 #1 SMP PREEMPT Sat Oct 19 19:58:02 UTC 2024 aarch64 Toybox"` **(Huawei, KIW-L21, 29) LineageOS:** `"Linux localhost 3.10.108-lineageos-g6f027fb1e3f #1 SMP PREEMPT Fri Nov 10 19:48:05 UTC 2023 aarch64"`
Memory Usable By Apps (MB) (Total*Swap)	**(Google, Pixel 6, 34):** `"7615 x 3071"`, `"5601 x 3071"`, `"7607 x 3071"` **(Realme, RMX3242, 33):** `"3615 x 5631"` **(Xiaomi, 2203129G, 34):** `"7177 x 6143"`
Device Ringtone	**Media Store internal:** `"content://media/internal/audio/media/204?title=Orion&canonical=1"` **Media Store external:** `"content://0@media/external/audio/media/1000000101?title=Mission%20 (Impossible%20Ti%C3%ABsto%20Remix)%20Radio%2016%20Bit%20MASTER&canonical=1"` **Theme Manager app:** `"content://0@com.android.thememanager.fileprovider/external-files-MIUI/ .ringtone/Nokia-Tone.mp3"` **System internal media:** `"file:///system/media/audio/ringtones/MiRemix.ogg"`
Communication Device Name	**(Xiaomi, 2203129G, 34) Bluetooth Headset:** `"JBL LIVE460NC"` **(Samsung, SM-G781V, 31) Wireless Bluetooth Earbuds:** `"Benjamin's Buds2 Pro"` **(Xiaomi, M2007J3SY, 31) Hearing Aid Device:** `"Aide aud. de ribeiro"`
Boot Count	**API 33:** `243`, **API 34:** `1606`
Network and SIM Operators	`("PROMOBILE SN", "Orange")`, `("AT&T", "AT&T")`, `("Turk Telekom", "Turk Telekom")`, `("ALG Mobilis", "mobilis")`, `("NL KPN", "Lebara")`, `("Lebara", "Lebara")`
Next Alarm Time	`"Fri 16:27"`(english), `"ven. 06:53"`(french), `"tors 06:00"`(Swedish), `"So., 12:30"`(German)