# Post-GDPR Threat Hunting on Android Phones: Dissecting OS-level Safeguards of User-unresettable Identifiers

Mark Huasong Meng[*§‡], Qing Zhang[†], Guangshuai Xia[†], Yuwei Zheng[†], Yanjun Zhang[¶§],
Guangdong Bai[§✉], Zhi Liu[†], Sin G. Teo[‡], Jin Song Dong[*]

[*]National University of Singapore, [†]ByteDance, [¶]Deakin University, Australia,
[§]The University of Queensland, Australia, [‡]Institute for Infocomm Research, A*STAR, Singapore
{menghs, teo_sin_gee}@i2r.a-star.edu.sg, g.bai@uq.edu.au, dcsdjs@nus.edu.sg

*Abstract*—Ever since its genesis, Android has enabled apps to access data and services on mobile devices. This however involves a wide variety of user-unresettable identifiers (UUIs), e.g., the MAC address, which are associated with a device permanently. Given their privacy sensitivity, Android has tightened its UUI access policy since its version 10, in response to the increasingly strict privacy protection regulations around the world. Non-system apps are restricted from accessing them and are required to use user-resettable alternatives such as advertising IDs.

In this work, we conduct a systematic study on the effectiveness of the UUI safeguards on Android phones including both Android Open Source Project (AOSP) and Original Equipment Manufacturer (OEM) phones. To facilitate our large-scale study, we propose a set of analysis techniques that discover and assess UUI access channels. Our approach features a hybrid analysis that consists of static program analysis of Android Framework and forensic analysis of OS images to uncover access channels. These channels are then tested with differential analysis to identify weaknesses that open any attacking opportunity. We have conducted a vulnerability assessment on 13 popular phones of 9 major manufacturers, most of which are top-selling and installed with the recent Android versions. Our study reveals that UUI mishandling pervasively exists, evidenced by 51 unique vulnerabilities found (8 listed by CVE). Our work unveils the *status quo* of the UUI protection in Android phones, complementing the existing studies that mainly focus on apps' UUI harvesting behaviors. Our findings should raise an alert to phone manufacturers and would encourage policymakers to further extend the scope of regulations with device-level data protection.

## I. INTRODUCTION

It is well known that many Android apps collect personally identifiable information (PII) from mobile devices to track users for the claimed purpose of enhancing user experience and personalization [45], [51], [57], [84]. Many types of PII are user-unresettable, such as the device ID and SIM card information, and thus we refer to them as *user-unresettable identifiers* (UUIs). They are not as easily replaceable as a password once leaked, so they deserve strict protection. Nevertheless, there is no way for the users to restrict how the apps use the collected UUIs. Additionally, a few popular analytics or advertisement libraries are so extensively reused among apps that cross-app behavior tracking and user profiling become viable [5], [7], [11], [55], [68], [75], [76], [87] through UUIs.

The protection of personal data has gained a great deal of attention around the world. Many countries have put in place legislation to regulate the collection and use of personal data [15], [16], [17], [72], such as the well-known European Union (EU) General Data Protection Regulation (GDPR) [70]. They impose strict obligations on data controllers. Infringements of user privacy could result in large penalties, e.g., *"a fine of up to €20 million, or 4% of the firm's worldwide annual revenue"* set by GDPR [78]. Google recently also has taken steps to enforce new privacy features to restrict apps' use of user data, especially on the UUIs [27], [28], [29]. Since Android 10, most UUIs are regulated with a new privileged permission of READ_PRIVILEGED_PHONE_STATE, which is only granted to privileged system apps [26] and the apps signed with the platform key[1]. Desensitized, app-unique and user-resettable identifiers (e.g., Android advertising IDs [25]) are introduced, and apps are required to use them whenever an identifier is in need. App developers are also required to release a privacy policy to disclose their apps' access, collection, use, and sharing of user data. They are obligated to engage proper mechanisms for securely handling data within the disclosed purposes [21].

The data exfiltration in Android has been extensively studied by the research community. Many efforts have been made to analyze data harvesting behaviors of third-party apps and their consequences through a variety of techniques such as program analysis [3], [14], [56], [83] and traffic analysis [2], [9], [77]. However, these offer only a partial view of the UUI exfiltration in the entire Android ecosystem. The complementary question of *whether the operating systems (OSes) themselves comprehensively safeguard UUIs* still remains open. Existing related research mainly focuses on identifying side

---

[1]Both are referred to as *system apps* hereafter. In contrast, we use *non-system apps* to denote the apps whose protectionLevel is normal or dangerous. In the remaining of this paper, *apps* indicate only non-system apps unless stated otherwise.

channels and covert channels [50], [58], [66], but there is still a lack of systematic and rigorous assessment of OS-level protection. Researchers may have simply assumed the OSes are able to securely manage the UUIs as they claim, particularly after a decade's advancement of Android. Indeed, some Android phones even have been certified with reputable standards before they are put on the market [8], [43], [61], [62].

In this work, we bridge this gap by conducting a comprehensive analysis of Android's OS-level enforcement of UUI protection. We first review Android's updates on its privacy policy in its two most up-to-date versions (10 and 11) as of December 2021, and we have identified six types of UUIs that have been stipulated by Google as restricted or inaccessible by non-system apps. We assess the *documented interfaces* that Android exposes for apps to access them, using a "regression testing" paradigm that invokes the APIs following the manners documented in the versions prior to Android 10 (i.e., API level 29). A non-regression, which occurs if a UUI is still accessible through any API, is reported as a failure.

Our approach also considers *undocumented interfaces*. It takes the six UUIs as seeds to discover software components and files (referred to as *access channels*) that UUIs may flow through, guided by the intuition that these channels are likely to engage other unknown UUI types and OEM-customized UUIs as well. In particular, it takes the APIs used to access the six UUIs as entry points and traces the call chain within the Android Framework using *static analysis*. The publicly accessible functions in the call stack are grouped for identifying the components they belong to. It also searches the dumped device images for the files hosting any of the six UUIs, and links them to software modules that manage them. Our approach manages to identify three categories of undocumented access channels, including *system services*, *system properties*, and *system settings*. The former represents various modules that serve APIs such as the `TelephonyManager`, while the latter two cover the main mechanisms that Android uses to share runtime and non-dynamic global device information.

The identified access channels are then assessed for weaknesses. The first challenge we overcome is to discover all app-accessible interfaces exposed by them. For the system services, we employ a hacking way to enumerate all methods in the service under testing. We use Java reflection to get the *binder* stub object and send it *transactions* containing method code, which is a short integer (so it is enumerable), via Android's binder interprocess communication (IPC). For the system settings and system properties, we obtain all keys via the Android Debug Bridge (ADB) debugger and enumerate them during our testing. The second challenge is to recognize UUIs from the obtained values, mainly unknown and OEM-defined UUIs. We use *differential analysis* to address this. Intuitively, our testing is repeated after rebooting and factory resetting the phone, and across phones of the same model. Only those device-unique values that have a non-short size and remain stable after rebooting and factory resetting are reported.

We apply our approach to assess phones of 13 models from 9 manufacturers. They are installed with the official AOSP or an OEM Android that is supposed to inherit the strict privacy policies and restrictions of AOSP. They all are the latest models

of their manufacturers and represent over 87% of the global market share of Android devices [67] as of April 2022.

**Key findings**. Our study is the first to rigorously investigate the OS-level UUI protection in Android. We unveil the *status quo* of this critical safeguard. Our key findings are summarized below.

- **Landscape of OS-level UUI safeguards.** The UUI mishandling issues are pervasive in the latest Android phones. Our study finds a total of 51 unique vulnerabilities, leading to 65 occurrences of UUI leakages. Almost every phone except AOSP 11 contains at least one UUI leakage, with the highest 10, mean 6, and median 5. AOSP is less susceptible to this issue than OEM OSes. Nonetheless, even though Google has proclaimed that the access to UUIs is tightened, we have found one non-trivial vulnerability from AOSP, which fails to be fixed due to incompatibility issue after Google has acknowledged it.
- **Exfiltration points.** The undocumented access channels are the major exfiltration points (45 out of all 51 vulnerabilities). Among them, 5 are caught from the system services, 10 in system settings, and the remaining 30 from system properties.
- **Whitelisting issues.** We find an issue in the phones of three manufacturers who excessively use the whitelisting mechanism in regulating the invocations of sensitive APIs. Due to flawed identity validation, a malicious app can trick the whitelisting mechanism and circumvent permission control to collect UUIs. It is the first time that such issues have been revealed to the public.
- **Exploits in the wild.** The broad UUI collection by apps in the wild is not observed in our study, but the identified channels have been (ab)used by a few popular apps from major app stores. We find that 12 out of analyzed 300 apps have relevant behaviors. All their accesses to UUIs are through undocumented access channels.

**Contributions**. The main contributions of this work are summarized as follows:

- **Understanding OS-level UUI protection.** We conduct the first comprehensive study on the Android OS-level UUI protection. Our work complements existing studies on app-level data harvesting behaviors, completing the PII protection research in the entire Android ecosystem.
- **A systematic assessment approach.** We propose a set of analysis techniques to automatically discover and assess UUI access channels on Android phones. Our approach features a hybrid analysis and applies to a broader range of Android devices and future versions of Android.
- **Revealing the *status quo* of UUI protection.** We present the landscape of UUI safeguards in the latest popular Android phones from major manufacturers. Our work should raise an alert to the manufacturers and encourage policymakers to further extend the scope of regulations with device-level data protection.

## II. BACKGROUND

This section introduces relevant background of Android's permission control, and the implication of the emerging personal data protection legislation to the Android ecosystem.

### A. Android Permission System

Android enforces strong isolation on its apps with the security mechanisms inherited from Linux, such as Linux user identifier (UID) and SELinux. The execution and data storage of the apps are strictly sandboxed. It employs a *permission-based* mechanism to regulate the resource access by the apps. All data and services on a device are protected with permission labels, and only the apps that are granted with correct permissions can access them.

The typical permission labels are grouped into three protection levels according to their sensitivity [33]. The lowest level is the *normal* protection level, and permissions at this level are automatically granted to the apps at the installation time. Many essential but insensitive permissions are classified into this level, such as wallpaper setting and vibration. A higher protection level is the *dangerous* level (also known as the *runtime* permissions). To access any API protected by dangerous-level permissions, the app must have properly requested the access in its manifest file and gained the user's explicit consent at runtime. Typical dangerous permissions include accessing photos and geographic location. Non-system apps can at most obtain permissions at the dangerous level. The remaining protection level is the *signature* level, which is designed for privileged operations. Permissions at this level can only be granted to apps developed by the entity that defines them. Therefore, only system apps, such as the settings app and the camera app, can request signature-level permissions pre-defined by the device manufacturer.

### B. Android's Response to Data Regulations

In recent years, many governments around the world have issued relevant laws and regulations to enforce the protection of user privacy. The EU GDPR [16] is a well-known example. It comes with a comprehensive regulation over the entire life-cycle of personal data, ranging from the access and collection of those data, their cross-border transmission, until the deletion. It ensures users with *the right to be informed* and *the right to consent*. Together with another EU law titled the Privacy and Electronic Communications Directive (better known as ePrivacy Directive) [15], it enforces that any information that facilitates in identifying a user is not allowed to be collected without the user's explicit consent.

In response to these regulations, mobile device manufacturers have started taking action. For example, the industry standards of mobile user privacy protection are made by the association of several device manufacturers [41]. Android also imposes strict privacy policies to regulate apps in handling user data, as Google's strategic response to the ePrivacy Directive and the GDPR [48]. Below, we present a brief review of the evolution of its privacy policies.

In its early versions, apps are provided with two convenient ways to obtain information that can identify the device. First, they can request an *Android ID* (i.e., the `Settings.Secure.ANDROID_ID`, or the SSAID), which is generated based on the device hardware ID and the user ID. Second, they can request permissions, which are primarily in the dangerous protection level, to read the device or system status, including the serial number and the MAC address.

After Google rolled out a series of privacy rules, these two ways have been disabled. Since Android 8, the public key of the app developer is incorporated in the Android ID generation so that the Android ID becomes app-unique. This prevents cross-app user tracking, unless among apps of the same developer. Android also introduces an alternative called *Android advertising ID* [25] to facilitate apps with advertisement needs. It can identify a specific user on a particular device, but unlike UUIs, the user is free to reset it. Google has escalated the permission of accessing UUIs since Android 10. Most UUIs are regulated with a new privileged permission `READ_PRIVILEGED_PHONE_STATE`, which is only granted to system apps, indicating that the access of non-system apps is revoked. In Section III-B, we present more details on the UUI access policies.

### III. Understanding Android UUIs

Although the data collection on Android has been extensively studied [58], [60], [64], [90], there still lacks a comprehensive understanding of Android UUIs. In this section, we present our definition of UUIs (Section III-A). We then summarize the UUIs recognized in Android documents and widely discussed in the literature (Section III-B), and characterize them (Section III-C).

### A. Definitions and Scope

We consider the information that can be used to directly or indirectly identify a device as identifiers. We refer to an identifier that has permanent binding with a device, e.g., the hardware equipment ID, as a *user-unresettable identifier* (UUI). They require complex processes, if not impossible, for the users to reset or revoke, and thus are termed user-unresettable.

### B. Recognizing UUIs

Since there is a lack of a comprehensive list of Android UUIs as the target of our study, we attempt to recognize some to guide the design of our assessment approach. We resort to two sources for this purpose, and we have identified six types as listed in column 2 of Table I. In the following, we present these two sources, defering the discussion on the identified UUIs in Section III-C.

**Android's official documentation**. When new versions (10 and 11) of Android are released, their privacy updates are disclosed in the developer documentation (see [28] for Android 11 and [27] for Android 10). These updates are introduced for the purpose of fixing security and privacy vulnerabilities, or complying with the data protection regulations such as the GDPR. They indicate the data that have raised the concerns of users, Android developers and policymakers, and thus become an ideal source.

We examine the documented updates to identify those identifiable and unresettable items, with a focus of two types. The first type includes the items that any privacy feature is designated to protect. For example, Android introduces MAC address randomization in Android 10 and requires the signature-level permission `NETWORK_SETTINGS` to disable it [29], to prevent apps from obtaining the real MAC address,

TABLE I: List of recognized Android UUIs

| No. | UUI | Category | Permission Updates across Android Versions† | Literature | | Validity in Android 11 |
|---|---|---|---|---|---|---|
| | | | | Considered Sensitive in | Collection Techniques Used | |
| 1 | Serial number | Chip & Cellular | v8-9: `READ_PHONE_STATE` required. ≥v10: `READ_PRIVILEGED_PHONE_STATE` required. | [57], [59], [68], [69] | Documented APIs: [68] System services: [69] | ✗ (since v10) ✓ |
| 2 | Device ID (IMEI or MEID) | | <v10: `READ_PHONE_STATE` required. ≥v10: `READ_PRIVILEGED_PHONE_STATE` required. | [7], [14], [20], [37], [46], [54], [56], [57], [58], [59], [60], [68], [69], [74], [77] | Documented APIs: [14], [20], [56], [58], [68] System properties: [7] System services: [69] | ✗ (since v10) ✓ ✓ |
| 3 | ICCID | | <v10: `READ_PHONE_STATE` required. ≥v10: `READ_PRIVILEGED_PHONE_STATE` required. | [14], [46], [54], [56], [57], [59], [60], [69], [74], [77] | Documented APIs: [14], [56] System services: [69] | ✗ (since v10) ✓ |
| 4 | IMSI | | <v10: `READ_PHONE_STATE` required. ≥v10: `READ_PRIVILEGED_PHONE_STATE` required. | [14], [20], [46], [54], [56], [57], [58], [59], [60], [69], [77] | Documented APIs: [14], [20] System services: [69] | ✗ (since v10) ✓ |
| 5 | Bluetooth MAC address | Wireless Module | All versions: `BLUETOOTH` required. ≥v6: Randomization or a fixed return value required. v6-10: `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` required. ≥v10: `ACCESS_FINE_LOCATION` becomes mandatory | [68] | Documented APIs: [68] | ✗ (since v6) |
| 6 | WiFi MAC address | | All versions: `ACCESS_WIFI_STATE` required. v6-9: Randomization suggested. ≥v10: Randomization becomes mandatory. | [7], [20], [37], [54], [57], [58], [59], [60], [63], [74], [77] | ioctl usage: [58] Documented APIs: [20], [63], [68] Read from /sys/class/net: [63] | ✗ (since v6) ✗ (since v10) ✗ (since v7) |

† The color schemes indicate the permission protection levels: *normal*, *dangerous* and *signature*.

so we identify it as a UUI. In this way, we have recognized UUIs #5 and #6 (column 1 of Table I).

The second type includes the items that are involved in any update on their permission levels or are newly taken into Android's permission control. For example, we include the IMEI because the permission requirement for the IMEI is escalated from the `READ_PHONE_STATE` permission to `READ_PRIVILEGED_PHONE_STATE` since Android 10, where the former can be granted to a non-system app, but the latter is limited to system apps only. In this way, we have identified UUIs #1-5.

**Literature**. Besides Android official documentation, we also refer to the literature for other UUI types. Since the UUIs have not been widely discussed, we expand our scope to the literature on Android data harvesting. We start with Wang *et al.* [74] and Shen *et al.* [64] that are the most recent publications in this area, and then track other publications they cite. The scope of referenced literature covers personal information gathering [7], [46], [54], [59], [60], [74], user tracking [12], [13], [52], [90], log and traffic monitoring [57], [64], [68], [77], covert/side channels [6], [58], [79], [86] and various app analysis techniques [14], [20], [56], [69]. We examine the data types that are considered sensitive by these studies and include only those unresettable items in our list. This helps us find the UUIs that concern our research community.

We summarize the UUIs identified from the literature in column 5 of Table I. They cover all types we have recognized from Android documentation. Besides the UUI types, we also have reviewed the UUI collection techniques proposed in the literature (column 6) and assessed their validity in the latest versions (column 7). We note that most of them have been fixed or disabled with the evolution of Android (marked as ✗ in column 7).

### C. Characterizing Android UUIs

The six recognized UUIs can be classified into two categories based on the components they are associated with (column 3 of Table I). In the following, we briefly describe them.

**Chip and cellular identifiers (4 UUIs)**. A phone has several unique alphanumeric codes for the purposes of identifying the device or establishing cellular communication with the base station.

*Serial numbers* (UUI #1 in Table I). Each phone is manufactured with a *serial number*, which is used for the manufacturers to trace their products during the manufacturing process and post-sale warranty service.

*Device IDs* (UUI #2). Each phone equipped with a cellular module owns a *device ID*. It is a fixed-length decimal digits formatted by authorities or cellular service providers. For the devices registered to use the Global System for Mobile Communication (GSM) service, the device ID is the *International Mobile Equipment Identity (IMEI)*, while for Code Division Multiple Access (CDMA) devices, it is the *Mobile Equipment Identifier (MEID)*.

*SIM card IDs*. SIM card IDs include the *Integrated Circuit Card ID (ICCID*, UUI #3*)* and the *International Mobile Subscriber Identity (IMSI*, UUI #4*)*.

These four UUIs have been considered highly sensitive in the latest Android versions. Prior to Android 10, an app can access them if it is granted a dangerous-level permission named `READ_PHONE_STATE`. Since Android 10, the access to them have been restricted to only system apps.

**Wireless module identifiers (2 UUIs)**. Network communication through WiFi and Bluetooth has been incorporated by almost all Android phones. Two UUIs, i.e., the *WiFi media access control (MAC) address* and the *Bluetooth MAC address* are used to identify a device during communication. Android has enforced strict regulations on both of them since Android 10 (see Table I). Bluetooth and WiFi MAC addresses can only be returned after randomization or masking with an all-0 string.

### IV. OVERVIEW OF OUR APPROACH

After recognizing the six UUIs, we design and implement U2-I2 (short for <u>U</u>UI <u>i</u>nvestigator)[2] which systematically investigates the OS-level UUI protection at large scale. We note

---

[2]U2-I2 is available at https://uq-trust-lab.github.io/u2i2/.

that U2-I2 is designed to assess the protection of not only the six known UUIs, but also other previously unreported ones (detailed in Section VI).

### A. Objectives and Threat Model

U2-I2 aims to 1) detect programmable interfaces through which a malicious non-system app can access a UUI without the required permissions, 2) identify inconsistency between the *de facto* permissions imposed on a UUI and Android's documented policy, and 3) discover non-compliance of AOSP's privacy policy on the OEM OSes.

**Threat model**. We consider an attacker who aims to collect UUIs from the mobile device of the victim. The attacker's capabilities are defined as follows.

First, the attacker runs its UUI collector only as a *non-system* app on the victim's device, or as a software library imported by a non-system app.

Second, the attacker's app is granted the Internet permission, which is used only for transmitting the collected data. Other than that, it requires no permission when using undocumented channels (Section VI). It can be granted the permissions that are valid in an earlier version of Android when using the documented interfaces (Section V).

Third, the attacker is able to conduct reconnaissance, profiling and analysis *offline* on various Android devices and OSes. There is no restriction on this, meaning that the attacker can obtain static/runtime information of the devices, and pre-built OSes and firmware, such as the signatures of packages/classes/methods and file system structure, via rooting the device, reverse engineering the firmware, and debugging/testing the OSes.

### B. Challenges and Approach Overview

The core idea of U2-I2 is to identify the access channels through which non-system apps may obtain any UUI, generate test cases tailored for an arbitrary Android device, and comprehensively test them to pinpoint weaknesses that may lead to a UUI leakage. During the investigation, the following main challenges are alleviated.

***Challenge 1: Identifying undocumented access channels***. U2-I2 tests the APIs listed in Android documentation, as detailed in Section V. Besides that, there may exist undocumented interfaces due to the complexity of Android. OEM manufacturers may also introduce customized interfaces. Therefore, U2-I2 explores other possible unprivileged access channels. This is detailed in Section VI-A.

***Challenge 2: Automating assessment process***. Due to the fragmentation, the ways to access a certain UUI on Android devices may greatly differ from each other. As U2-I2 aims to comprehensively assess multiple access channels in AOSP and multiple heterogeneous OEM phones, its assessment process is automated as much as possible to make it scalable.

***Challenge 3: Pinpointing customized UUIs***. The six recognized UUIs may just be the tip of the iceberg. Device manufacturers may define new types of UUIs, which also put users' privacy at risk if they are not properly protected. Therefore, U2-I2 employs a differential testing to pinpoint

UUIs that are customized by device manufacturers yet to be widely recognized and studied. This is detailed in Section VI-B and VI-C.

### V. ASSESSING DOCUMENTED CHANNELS

Android developer documentation details the programmable interfaces to access data on the device. U2-I2 thus searches it for the APIs for retrieving the six known UUIs. It identifies nine distinct relevant APIs, which are listed in the first two columns of Table II. It turns out that for any of our recognized UUIs, Android provides at least one API in particular versions. The API for reading the serial number is provided by the `Build` class, while those chip and cellular-specific UUIs are mainly managed by the `TelephonyManager`. APIs for wireless module identifiers are distributed in three classes (i.e., `BluetoothAdapter`, `WifiManager` and `WifiInfo`).

U2-I2 builds a non-system app to test these interfaces. It takes the strategy of software *regression testing*. In particular, given that the vulnerabilities are more likely to be introduced during version upgrading or OS customization, the tester app includes the *cross-version validation* and the *cross-manufacturer validation*, which validate whether the security policies specified in the official documentation are complied with by various Android versions and various devices. During the testing, we check the following two types of errors.

**Legacy permissions**. As shown in column 5 of Table I, the imposed permission control on each UUI since Android 10 becomes disparate from that in the earlier versions. Some devices thus may fail to keep their permission controls up to date. To identify non-compliance of this type, U2-I2 grants the tester app either a permission valid for only earlier versions, or none of the required permissions, and then checks whether it can still obtain the UUI. For example, the `getSerial()` method is restricted to only system apps since Android 10, so any invocation from a non-system app, even with the `READ_PHONE_STATE` permission that is valid in version 8 and 9, should trigger a security exception in Android 10 or 11.

**Missing de-identification**. Android documentation also claims that some APIs return randomized UUIs to non-system apps, e.g., the Bluetooth MAC address. When testing these APIs, we pass the test harness (e.g., required permissions and genuine identifiers) to U2-I2. It grants the tester app required permissions and compares the return values with the genuine identifiers to validate whether the randomization is conducted on the UUIs it retrieves.

### VI. DISCOVERING AND ASSESSING UNDOCUMENTED ACCESS CHANNELS

U2-I2 proposes a three-phase approach to exploring and assessing undocumented access channels, as shown in Figure 1. In this section, we present each phase.

### A. Access Channel Exploration

The access channel exploration of U2-I2 (step ❶ in Figure 1) takes the six known UUIs as seeds, considering that other unknown UUIs may share the same set of access channels. It uses the following two techniques to explore undocumented access channels.
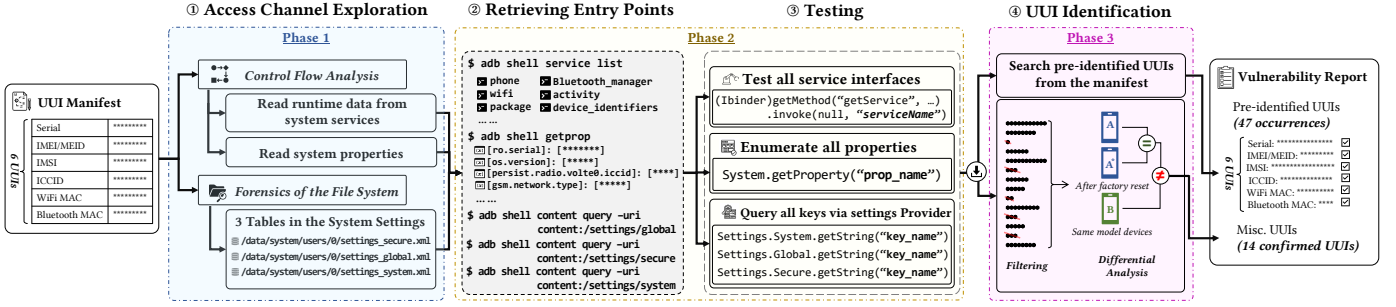
Fig. 1: The workflow of our UUI exploration and assessment through undocumented channels
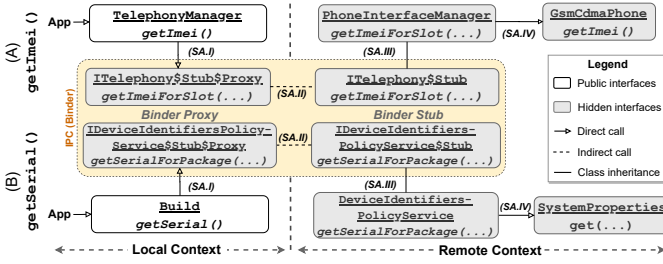


Fig. 2: Visualization of two typical control flows starting from documented APIs

*1) Static Control Flow Analysis:* Typically, when an Android app invokes an API in a service manager to access system sources, the invocation would reach a *local interface* in a Binder *proxy*. The proxy then sends the request to a *stub* through Android's inter-process communication (IPC) mechanism, i.e., the Binder. The stub eventually invokes a system service, which directly or indirectly (e.g., invoking another component) serves the request. Figure 2 illustrates this process with two typical control flows. Below we explain how we use a static control analysis to identify the components involved in serving UUI-related APIs.

First, we take an API (e.g., getImei()) as the entry point and extract the method-level call relations (SA.I in Figure 2) with the static analyzer soot [73]. It gives us a path ending at the local interface of the service manager (e.g., getImeiForSlot()), which is defined in a Binder proxy (e.g., ITelephony$Stub$Proxy).

Second, we map the local interface to the corresponding *remote interface* defined in a Binder stub class (SA.II). Both the proxy and stub inherit from the same interface class. For example, both ITelephony$Stub$Proxy (proxy) and ITelephony$Stub (stub) inherit from ITelephony. They are actually generated based on Android Interface Definition Language (AIDL), and thus, they have the same method signature (i.e., method names, argument number and types, and return types).

Third, we target to pinpoint the components the stub invokes to serve the request. We first traverse the inheritance of the remote interface to locate the method overriding the interface (SA.III) (e.g., getImeiForSlot() in PhoneInterfaceManager). We then extract the call stack starting with the method (SA.IV). By checking the involved classes and packages in the path, we find that the call paths are within

the system services in most cases. For example, in Figure 2-A), the path ends at getImei() in GsmCdmaPhone, which belongs to the system service phone. Thus, we treat *system services* as one of the access channels to assess in later phases.

Next, we examine the classes in the path that are out of system services to explore additional access channels. We find only one case of this type, as shown in Figure 2-B. The serial number is not directly read from a system service but is eventually queried from a system property. We thus treat *system properties* as the second type of access channels. They are used to store configuration and status information of hardware and operating system, and are initialized at device booting time [22]. A property is identified by its key (e.g., ro.serial for the serial number), and its value is stored inside the system directory.

We present more details on our control flow analysis on the two examples (Figure 2) in Appendix A. We note that we do not target building an exhaustive call graph since our study only aims to identify the components involved in UUI access. Accordingly, we adopt a coarse-grained strategy: 1) a call to a virtual method is resolved to call any of its subclasses, and 2) a call to an interface is resolved to call any class implementing it. Moreover, we only focus on the Java context in step SA.IV as none of the APIs in Table II calls into a native system service (further discussed in Section IX).

*2) Forensics of the File Systems:* Second, U2-I2 searches for the values of the six UUIs in the dumped images of devices. It excludes the directories that have been strictly restricted by Android OS (e.g., /proc) and exclusive folders owned by system apps (e.g., /sys). It catches the occurrence of some UUIs (e.g., Bluetooth MAC addresses) from the .xml files inside the /data/system/users/ directory. There are three xml files located in that directory, namely Settings.Secure, Settings.Global and Settings.System, which are altogether known as the system settings. These settings act as key-value databases to store a list of system-level device preferences, which are defined by the OS and system apps. Therefore, U2-I2 recognizes *system settings* as the third access channel.

### B. Automatic Access Channel Testing

With the three undocumented access channels identified, U2-I2 proceeds with testing them. Given an Android phone, it first retrieves all *entry points* in each access channel (step ❷ in Figure 1) and then enumerates them during testing (step ❸).

TABLE II: Documented APIs to access six known UUIs and corresponding system services

| UUI | Developers API Name(s) | System Service(s) |
|---|---|---|
| Serial | `android.os.Build.getSerial()` | `device_identifiers` |
| Device ID/IMEI/MEID | `android.telephony.TelephonyManager.getImei()` | `phone` |
| | `android.telephony.TelephonyManager.getDeviceId()` | `phone` |
| | `android.telephony.TelephonyManager.getMeid()` | `phone` |
| ICCID | `android.telephony.TelephonyManager.getSimSerialNumber()` | `phone` |
| | `android.telephony.SubscriptionInfo.getIccId()`[†] | `isub` |
| IMSI | `android.telephony.TelephonyManager.getSubscriberId()` | `phone` |
| Bluetooth MAC | `android.bluetooth.BluetoothAdapter.getAddress()` | `bluetooth_manager` |
| WiFi MAC | `android.net.wifi.WifiInfo.getMacAddress()`[‡] | `wifi` |

[†] To invoke that API, the app must obtain an instance of *SubscriptionInfo* through another API `SubscriptionManager.getActiveSubscriptionInfo()` at first.
[‡] To invoke that API, the app must obtain an instance of *ConnectionInfo* through another API `WifiManager.getConnectionInfo()` at first.

*1) Retrieving Entry Points:* U2-I2 leverages the Android Debug Bridge (ADB) to identify entry points from the three access channels. For system services, it uses `adb shell service list` command to retrieve the full service list and the package names of all services on the phone. For system settings, it uses `adb shell content query` to retrieve the keys of all stored entries, e.g., `System.sim1_imsi`. For system properties, it uses `adb shell getprop` to retrieve the keys of all stored entries, e.g., `gsm.imei1`.

We remark that ADB is not the only way to retrieve entry points. In Appendix B, we list alternatives based on Java interfaces. It is also worth noting that the presented steps can be done *offline* by the attacker to obtain the entry points, and the attack app needs *no* permission to query the values, as we discuss next.

*2) Testing:* The actual testing of U2-I2 is through a tester app installed on the phones to assess. We define each execution of testing on a certain device as a *testing session*. For each model, we test at least two distinct phones with the same version of OS installed. Moreover, we also perform at least two testing sessions on the same phone, on the premise of conducting a factory reset over the device between different sessions. Thus, we can collect the outputs in both *cross-device* and *cross-session* manners for differential analysis (detailed soon in Section VI-C). Below we detail our testing for each access channel.

**System services**. The client and the server of an Android system service use the same AIDL interface to facilitate their RPCs. An `IBinder` interface is generated based on the AIDL interface, and the service implements it. In a normal access, i.e., when an app calls the `getSystemService()` method, the client binds the service and then invokes the methods from the returned `IBinder` object like normal procedure calls. Nonetheless, this requires the caller to have an application context, and the invocation is under the regulation of Android permission system. U2-I2 thus resorts to a "*hacking way*" through Java reflection to bypass the permission check by `getSystemService()`. In the following, we take the exploit code of accessing the serial number on one of our tested devices as an example, as listed in Figure 3, to explain our method.

First, U2-I2 calls the `getService()` method using Java reflection to obtain the `IBinder` object (line 3 of Figure 3). Here the service name, e.g., the "knoxcustom" in our example, is obtained during entry point retrieval (Section VI-B1). It then

```java
1  public static void getSerialNumber() {
2    try {
3      IBinder iBinder = (IBinder) Class.forName("android.os.
         ServiceManager").getMethod("getService",String.class)
         .invoke(null, "knoxcustom");
4      Parcel data = Parcel.obtain();
5      Parcel reply = Parcel.obtain();
6      data.writeInterfaceToken("com.samsung.android.knox.custom.
         IKnoxCustomManager");
7      if (iBinder.transact(195, data, reply, 0))
8        String serialNumber = reply.readString();
9      ...
10   } catch (Exception e) { ... }
11 }
```

Fig. 3: Accessing serial number through system service on one of the tested devices (exploitation of CVE-2021-25344)

specifies the service name and package name to construct the parcels (line 6). We note that Android has explicitly listed `getService()` as a non-SDK interface [31] and labeled it as `@UnsupportedAppUsage`. In Section IX-B, we discuss our recommendations on such non-SDK interfaces.

After obtaining the `IBinder` object, U2-I2 calls its `transact()` method (line 7), which matches `Binder`'s `onTransact()` interface. The latter then calls the service object on the server side through Android's binder mechanism. Here the first argument is a code which is an integer between 1 to $2^{24}$-1 that indicates the action to perform. For example, by reverse engineering the image of the device we find that 195 in our example indicates the `getSerialNum()` method of the `IKnoxCustomManager`. U2-I2 enumerates 1 to $2^{24}$-1 during testing so as to invoke every method defined in each bound service. We learn from the analysis in Section VI-A that the methods involved in system services are usually getter functions that request zero or limited number of simple parameters. Therefore, we test those methods with a set of predefined parameters in both primitive types and non-primitive types, including an integer 0 or 1 that indicates the SIM card slot index, and a string specifies the package name of our test app. Once a method successfully returns, U2-I2 captures the returned data (line 8) and passes them to the differential analysis (Section VI-C). In Appendix C, we detail this process.

**System properties** and **system settings.** A system property can be accessed by querying its key through `System. getProperty()`. U2-I2 enumerates all keys obtained during the entry point retrieval to fetch their values. For system

| # | UUI | Access Channels | Exfiltration Points (with Devices) | Total |
|---|-----|-----------------|------------------------------------|-------|
| 1 | Serial Number | Documented APIs | `android.os.Build.getSerial()` ($\underline{A}_{10}$) | 1 |
| | | System Properties | `gsm.sn` ($\underline{B}_9$), `persist.radio.serialno` ($\underline{C}_{10,11}$), `ril.serialnumber` ($\underline{E}_{10}$), `ro.ril.oem.psno` ($\underline{H}_{10,11}$), `ro.vendor.serialno` ($\underline{C}_{10}$), `ro.vendor.vold.serialno` ($\underline{D}_{11}$,$\underline{G}_{10}$), `ro.vold.serialno` ($\underline{D}_{10}$) | 7 |
| | | System Services | `knoxcustom.getSerialNumber()` ($\underline{E}_{10}$) | 1 |
| 2 | Device ID/IMEI/MEID | Documented APIs | `android.telephony.TelephonyManager.getImei()` ($\underline{A}_{10}$), `android.telephony.TelephonyManager.getDeviceId()` ($\underline{A}_{10}$), `android.telephony.TelephonyManager.getMeid()` ($\underline{A}_{10}$) | 3 |
| | | System Properties | `gsm.imei{0/1}` ($\underline{B}_9$), `gsm.meid` ($\underline{B}_9$), `persist.sys.show.device.imei` ($\underline{G}_{10}$), `ro.boot.deviceid` ($\underline{G}_{10}$), `ro.boot.em.did` ($\underline{E}_{10}$), `ro.rpmb.board` ($\underline{G}_{10}$) | 6 |
| | | System Services | `phone_huawei.getMeidForSubscriber(int slot)` ($\underline{A}_{10}$)†, `oiface.getDeviceId()` ($\underline{D}_{11}$) | 2 |
| 3 | ICCID | Documented APIs | `android.telephony.SubscriptionInfo.getIccId()` (AOSP 10,$\underline{A}_{10}$,$\underline{C}_{10}$,$\underline{D}_{10}$,$\underline{E}_{10}$,$\underline{F}_{10}$,$\underline{G}_{10}$,$\underline{H}_{10}$) | 1 |
| | | System Settings | `System.last_main_card_iccid` ($\underline{A}_{10}$), `Global.oppo_comm_simsettings_daily_alert_{iccid val}` ($\underline{D}_{11}$), `Global.color_data_roaming{iccid val}` ($\underline{D}_{11}$), `Global.volte_call_status{iccid val}` ($\underline{D}_{11}$) | 4 |
| | | System Properties | `persist.radio.volte0.iccid` ($\underline{G}_{10}$), `persist.radio.volte1.iccid` ($\underline{G}_{10}$), `persist.radio.ddssim.iccid` ($\underline{C}_{10}$) | 3 |
| 4 | IMSI | System Settings | `System.sim1_imsi` ($\underline{B}_9$), `System.sim2_imsi` ($\underline{B}_9$), `System.sim1_value` ($\underline{E}_{10}$) | 3 |
| | | System Services | `phone.vivoTelephonyApi(int slot)` ($\underline{G}_{10}$)† | 1 |
| 5 | Bluetooth MAC address | Documented APIs | `android.bluetooth.BluetoothAdapter.getAddress()` ($\underline{F}_{10}$) | 1 |
| | | System Properties | `sys.bt.address` ($\underline{B}_9$) | 1 |
| 6 | WiFi MAC address | System Properties | `persist.oppo.wlan.macaddress` ($\underline{D}_{10}$), `persist.sys.wififactorymac` ($\underline{G}_{10}$) | 2 |
| | | System Services | `wifi.getWifiFactoryMac()` ($\underline{G}_{10}$) | 1 |

† The method returns a UUI when it is tested with an integer parameter equals 0.

settings, their values can be accessed through the system settings provider (`android.provider.Settings`). U2-I2 queries all keys stored in all three tables. Similarly, all fetched values are then passed to the differential analysis.

### C. OEM-defined UUI Identification

In this phase, U2-I2 recognizes UUIs from the outputs of the testing. It first recognizes the values of the six known types, by matching the outputs with values fetched through documented methods. After that, the critical task is to pinpoint unknown or OEM-defined UUIs. To this end, U2-I2 takes the following two steps.

*Filtering.* U2-I2 first excludes those values of insufficient size, i.e., strings containing less than 4 hex-digit, as they cannot convincingly identify a device.

*Differential analysis.* After filtering, U2-I2 conducts differential analysis over the values. It excludes the values that are the same across devices, although they are in the same model and produced by the same manufacturer, as shown among devices **A** and **B** in step ❹. It keeps those that remain unchanged after factory-resetting the device, shown as devices **A** and **A\*** in step ❹. In the end, the remaining values are presented to the analyst.

## VII. EVALUATION AND LANDSCAPE OF UUI PROTECTION IN ANDROID PHONES

We apply U2-I2 on 13 popular Android smartphone models available in the market to understand the landscape of UUI safeguards in contemporary Android phones. The tested phones are from 9 distinct phone manufacturers, including *Google Pixel*, *Huawei*, *Lenovo*, *OnePlus*, *Oppo*, *Samsung*, *Smartisan*, *Vivo*, and *Xiaomi*. They together represent around 85% of the global market share of Android devices [67] as of December 2021. We note that our work aims to study the *status quo* in general, and it preserves manufacturer neutrality and avoids bias against certain device manufacturers. We assign each manufacturer except Google Pixel (which is installed

with the AOSP) with a code from $\underline{A}$ to $\underline{H}$ according to the aforementioned order throughout the paper. When referring to a device, we use the subscript number to denote the version of Android installed on it. For example, $\underline{E}$ stands for the manufacturer *Samsung*, and $\underline{E}_{10}$ stands for a *Samsung* device that is installed with Android 10.

Our investigation aims to answer the following three research questions (RQs).

**RQ1**. What is U2-I2's performance in identifying UUI mishandling issues? Based on U2-I2's findings, what is the *status quo* of UUI protection in the latest Android phones? Do their OSes comply with the up-to-date privacy policy of AOSP?
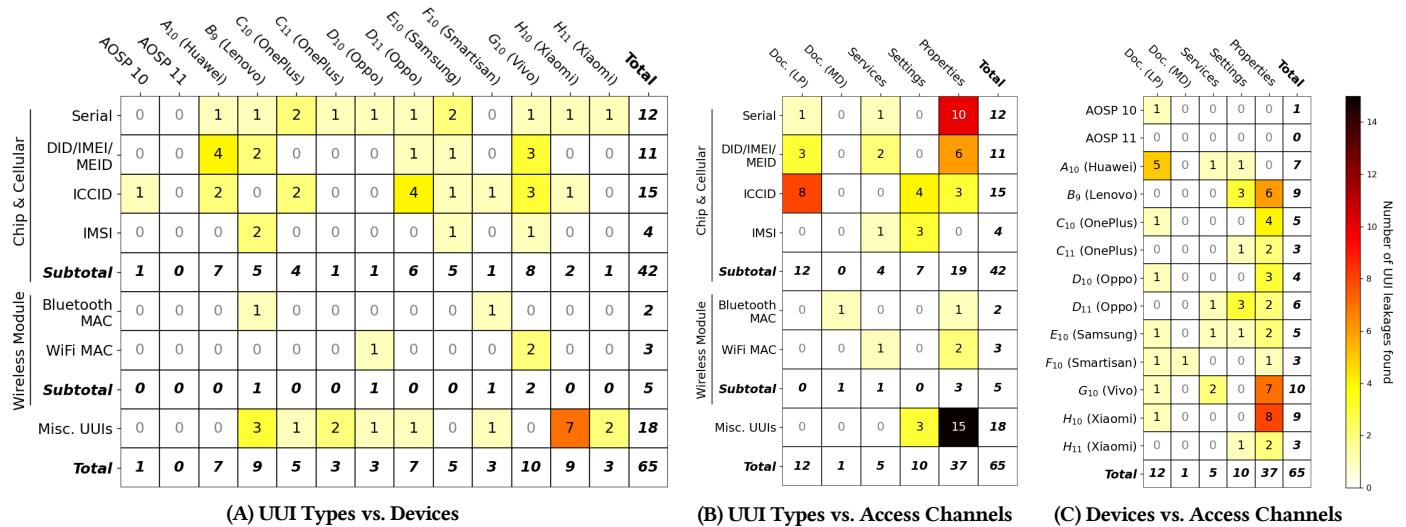
**RQ2**. Based on U2-I2's findings, what UUIs are potentially exposed to non-system apps, and what are the typical exfiltration points?

**RQ3**. Have our identified access channels of UUI access already been (ab)used by apps in the wild?

### A. RQ1: Status Quo *of OS-level UUI Protection*

U2-I2 has detected 51 unique vulnerabilities that lead to 65 occurrences of UUI leakage from the tested phones. It is capable of detecting vulnerable exfiltration points that involve the six known UUIs (listed in Table III), and those involving unknown or OEM-defined UUIs (listed in Table IV). We have reported them together with our suggested fixes to the relevant manufacturers. All of them have confirmed and acknowledged our findings, implying that U2-I2 generates no false positive.

While deferring the characterization of the identified vulnerabilities to Section VII-B, in the remaining of this section, we present the general picture on OS-level UUI protection in current Android OSes. Overall, almost every device (except AOSP 11) is found with at least one leakage, and most of them contain at least three, as shown in Figure 4-A, suggesting the pervasiveness of the UUI mishandling weaknesses in the latest Android phones.

**(A) UUI Types vs. Devices**

| | | AOSP 10 | AOSP 11 | $A_{10}$ (Huawei) | $B_9$ (Lenovo) | $C_{10}$ (OnePlus) | $C_{11}$ (OnePlus) | $D_{10}$ (Oppo) | $D_{11}$ (Oppo) | $E_{10}$ (Samsung) | $F_{10}$ (Smartisan) | $G_{10}$ (Vivo) | $H_{10}$ (Xiaomi) | $H_{11}$ (Xiaomi) | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Chip & Cellular | Serial | 0 | 0 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 0 | 1 | 1 | 1 | *12* |
| | DID/IMEI/MEID | 0 | 0 | 4 | 2 | 0 | 0 | 0 | 1 | 1 | 0 | 3 | 0 | 0 | *11* |
| | ICCID | 1 | 0 | 2 | 0 | 2 | 0 | 0 | 4 | 1 | 1 | 3 | 1 | 0 | *15* |
| | IMSI | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | *4* |
| | *Subtotal* | *1* | *0* | *7* | *5* | *4* | *1* | *1* | *6* | *5* | *1* | *8* | *2* | *1* | *42* |
| Wireless Module | Bluetooth MAC | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | *2* |
| | WiFi MAC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | *3* |
| | *Subtotal* | *0* | *0* | *0* | *1* | *0* | *0* | *1* | *0* | *0* | *1* | *2* | *0* | *0* | *5* |
| | Misc. UUIs | 0 | 0 | 0 | 3 | 1 | 2 | 1 | 1 | 0 | 1 | 0 | 7 | 2 | *18* |
| | *Total* | *1* | *0* | *7* | *9* | *5* | *3* | *3* | *7* | *5* | *3* | *10* | *9* | *3* | *65* |

**(B) UUI Types vs. Access Channels**

| | | Doc. (LP) | Doc. (MD) | Services | Settings | Properties | **Total** |
|---|---|---|---|---|---|---|---|
| Chip & Cellular | Serial | 1 | 0 | 1 | 0 | 10 | *12* |
| | DID/IMEI/MEID | 3 | 0 | 2 | 0 | 6 | *11* |
| | ICCID | 8 | 0 | 0 | 4 | 3 | *15* |
| | IMSI | 0 | 0 | 1 | 3 | 0 | *4* |
| | *Subtotal* | *12* | *0* | *4* | *7* | *19* | *42* |
| Wireless Module | Bluetooth MAC | 0 | 1 | 0 | 0 | 1 | *2* |
| | WiFi MAC | 0 | 0 | 1 | 0 | 2 | *3* |
| | *Subtotal* | *0* | *1* | *1* | *0* | *3* | *5* |
| | Misc. UUIs | 0 | 0 | 0 | 3 | 15 | *18* |
| | *Total* | *12* | *1* | *5* | *10* | *37* | *65* |

**(C) Devices vs. Access Channels**

| | Doc. (LP) | Doc. (MD) | Services | Settings | Properties | **Total** |
|---|---|---|---|---|---|---|
| AOSP 10 | 1 | 0 | 0 | 0 | 0 | *1* |
| AOSP 11 | 0 | 0 | 0 | 0 | 0 | *0* |
| $A_{10}$ (Huawei) | 5 | 0 | 1 | 1 | 0 | *7* |
| $B_9$ (Lenovo) | 0 | 0 | 0 | 3 | 6 | *9* |
| $C_{10}$ (OnePlus) | 1 | 0 | 0 | 0 | 4 | *5* |
| $C_{11}$ (OnePlus) | 0 | 0 | 0 | 1 | 2 | *3* |
| $D_{10}$ (Oppo) | 1 | 0 | 0 | 0 | 3 | *4* |
| $D_{11}$ (Oppo) | 0 | 0 | 1 | 3 | 2 | *6* |
| $E_{10}$ (Samsung) | 1 | 0 | 1 | 1 | 2 | *5* |
| $F_{10}$ (Smartisan) | 1 | 1 | 0 | 0 | 1 | *3* |
| $G_{10}$ (Vivo) | 1 | 0 | 2 | 0 | 7 | *10* |
| $H_{10}$ (Xiaomi) | 1 | 0 | 0 | 0 | 8 | *9* |
| $H_{11}$ (Xiaomi) | 0 | 0 | 0 | 1 | 2 | *3* |
| *Total* | *12* | *1* | *5* | *10* | *37* | *65* |

**Legend for documented channels:** Doc. (LP): Documented (Legacy Permission), Doc. (MD): Documented (Missing De-identification)

Fig. 4: Statistics of the occurrence of UUI leakages detected in our assessment, counted by UUI types and devices (A), by UUI types and access channels (B), and by devices and access channels (C)

TABLE IV: List of 14 miscellaneous UUIs found by U2-I2 (all acknowledged by manufacturers)

| Chl. | Name of UUI (with Devices) | Format & Purpose |
|---|---|---|
| System Settings | Global.cplc ($C_{11}$) | 45-byte hex string, for NFC module |
| | Global.ro.boot.oled_wp ($H_{11}$) | 8-byte hex string, for OLED display panel |
| | System.ReaperAssignedDeviceId ($B_9$) | 30-digit decimal string, unknown type$^\dagger$ |
| System Properties | gsm.serial ($C_{10,11}$,$D_{10,11}$) | 19-digit decimal string, the device PCB serial number |
| | ro.ril.oem.sno ($H_{10,11}$) | 8-byte hex string, unknown type$^\dagger$ |
| | vendor.camera.sensor.frontMain.fuseID ($H_{10}$) | 64-byte alphanumeric string, ID of the front main camera |
| | vendor.camera.sensor.rearMain.fuseID ($H_{10}$) | 64-byte alphanumeric string, ID of the rear main camera |
| | vendor.camera.sensor.rearUltra.fuseID ($H_{10}$) | 64-byte alphanumeric string, ID of the rear ultra-wide camera |
| | vendor.camera.sensor.rearTele.fuseID ($H_{10}$) | 64-byte alphanumeric string, ID of the rear telephoto camera |
| | persist.vendor.sys.fp.info ($H_{10}$) | 8-digit hex string, fingerprint sensor related, detail not found |
| | persist.vendor.sys.fp.uid ($H_{10}$) | 14-digit hex string, fingerprint sensor related, detail not found |
| | ro.qchip.serialno ($F_{10}$) | 8-digit hex string, a serial number of an embedded chip module |
| | ro.recovery_id ($B_9$) | 32-digit hex string, ID of the boot image |
| | ro.expect.recovery_id ($B_9$) | 32-digit hex string, a same value as above |

$^\dagger$ The UUIs labelled as "unknown type" contain insufficient information in their names and values.

**AOSP**. The UUI mishandling appears less severe in official AOSP than in the OEM OSes. The Google Pixel phones contain the smallest number of vulnerabilities among all devices, with only one found in AOSP 10 (i.e., the CVE-2021-0428[3]).

In AOSP 10, we find that one of the two APIs for the SIM card ID, i.e., the `getIccid()` in the class `SubscriptionInfo`, keeps returning the actual ICCID. This violates Android's privacy policy that accesses to the ICCID are prohibited to all non-system apps [27]. Nonetheless, the other

API for the ICCID, i.e., the `getSimSerialNumber()` in the `TelephonyManager`, has been updated in AOSP 10 to comply with the policy. We therefore speculate that the developers may have missed updating the rarely used `SubscriptionInfo` class, while they mainly focused on the `TelephonyManager` class which provides getters for most hardware IDs.

The fixing of this vulnerability turns out to be an unexpectedly non-trivial process. During our assessment, we notice that the vulnerability in `getIccid()` only exists in AOSP 10 and is not inherited by AOSP 11 and later releases. This may be because Google has been aware of this issue and has fixed it in AOSP 11, or it may have unintentionally fixed that vulnerability during the development process. After we reported it, Google issued a warning in its security bulletin on 5th Apr 2021 [24], which confirms the issue has been fixed. However, later in June 2021, the "fix" was recalled as shown in an updated version of the security bulletin. Google explained that the recall was because of "*application compatibility issues*". They later updated the comments in the source code of `getIccid()` [23], stating that the system-level permission for this method is imposed since Android 11 (API level 30) rather than Android 10 (API level 29). This, however, contradicts Android's documentation and the policies on other UUIs, where the system-level permission is imposed since Android 10.

**OEM Android**. The OEM OSes have more UUI mishandling issues than AOSP. This is somewhat expected, given the notorious fragmentation issue of the Android ecosystem [91]. Unsurprisingly, those installed OSes based on Android 10 have inherited the vulnerability from AOSP (i.e., the CVE-2021-0428), suggesting that the manufacturers may have never undertaken a compliance checking against Android documented policy when they conduct major OS updates.

On the documented APIs, most of them have well followed AOSP's update. Two exceptions appear in the permission

---

[3]The details of CVE-2021-0428 and Google's fix are available at https://uq-trust-lab.github.io/u2i2/cve/.

regulation and randomization though. *Huawei* (<u>A</u>) mis-exposes UUIs in its APIs for serial numbers and device IDs, and *Smartisan* (<u>F</u>) misses randomization on the returned Bluetooth MAC address.

However, the OEM OSes are found largely fail to ensure sufficient protection in their customized components. As shown in Table III, they commonly store device-specific values into the system settings and the system properties. This practice may be convenient for sharing data among their system apps but violates the privacy policy of AOSP and allows malicious apps to collect the stored UUIs. Similarly, they also fail to regulate the customized service managers, as shown in Table III.

In addition, we find the devices manufactured by *Huawei* (<u>A</u>$_{10}$), *Oppo* (<u>D</u>$_{10,11}$) and *Vivo* (<u>G</u>$_{10}$) whitelist apps for UUI access, and their whitelisting mechanisms have security risks. By exploiting them, the malicious apps can circumvent Android's permission control and access various UUIs without the required permission. We present details of these issues in Section VIII.

> **Findings in RQ1:** *U2-I2 is capable of detecting leakages of both known UUIs and OEM-customized UUIs. Our study finds that UUI mishandling is pervasive in popular Android phones. From the tested devices in 13 models of 9 manufacturers, 51 vulnerabilities are identified. They lead to 65 UUI leakages. Each tested device except AOSP 11 has been caught at least one leakage, with the highest 10, mean 6 and median 5.*
>
> *OEM OSes have more issues than AOSP. The main causes lie in the undocumented access channels. Three OEM OSes whitelist apps for UUI access, and security risks are found from their whitelisting mechanisms.*

**Responsible disclosure**. We have reported all our findings (including the whitelisting issues) to the relevant parties. We also have kept them confidential for at least 90 days for them to be mended before we reported them in this paper. As those eight issues on the `getICCID()` are caused by the vulnerability in AOSP, we report them to Google so that the affected manufacturers could receive timely warnings through Google's bulletins. For the remaining issues, we report them to the corresponding manufacturers.

All of them have acknowledged our findings, although a couple of manufacturers rate the reported issues with low severity. Most of them have been active and kept us updated on their fixing processes. We have received 8 common vulnerabilities & exposures (CVE) entries registered by Google and four other manufacturers, as listed in Table V.

### B. RQ2: Characterization of UUI Leakage Vulnerabilities

In this section, we characterize the identified vulnerabilities, with a focus on their distribution among UUI types and exfiltration points.

**Vulnerability distribution among UUI types**. All six known UUIs are found possible to be leaked in at least one device, as shown in Figure 4-A. In addition to them, we identify 14 new types of UUIs which are recognized because they keep

TABLE V: All CVE-listed vulnerabilities found by U2-I2

| # | CVE ID | Affected Devices | Involved UUIs |
|---|--------|------------------|---------------|
| 1 | CVE-2020-12488 | <u>G</u>$_{10}$ | Serial |
| 2 | CVE-2020-14103 | <u>H</u>$_{10,11}$ | Serial |
| 3 | CVE-2020-14105 | <u>H</u>$_{10,11}$ | Misc. UUID (sno) |
| 4 | CVE-2021-0428 | AOSP 10 | ICCID |
| 5 | CVE-2021-25344 | <u>E</u>$_{10}$ | Serial |
| 6 | CVE-2021-25358 | <u>E</u>$_{10}$ | IMSI |
| 7 | CVE-2021-26278 | <u>G</u>$_{10}$ | WiFi MAC address |
| 8 | CVE-2021-37055 | <u>A</u>$_{10}$ | ICCID |

constant after we factory reset the phones (see our approach in Section VI-C). We refer to them as the *miscellaneous UUIs*. They are introduced by five manufacturers (*Lenovo*, *OnePlus*, *Oppo*, *Smartisan* and *Xiaomi*), and all are found from their system settings and system properties (Table IV). We attempt to interpret these UUIs through their names and search online for details about them. Ten of them are identities of embedded equipment, such as the camera, the fingerprint sensor, the NFC module, the PCB (printed circuit board), and the screen panel; two are the identifies of the boot image; the types of the remaining two are unknown.

The ICCID (15), the serial number (12) and the device ID (11) are the top three exposed UUIs. Among the 15 ICCID leakages, over half (8/15) are caused by the AOSP vulnerability in the API. The other seven are through system settings or system properties. The vast majority (10/12) of the serial number leakages are through system properties. Over half of the device ID leakages (6/11) are through the system properties. All three top exposed UUIs are in the chip and cellular category.

**Leakage channels**. Figure 4-B summarizes the leakage distribution among the access channels. The documented APIs turn out to be relatively secure, although the vulnerability in AOSP 10 (i.e., CVE-2021-0428) affects all phones that are based on Android 10. Other than that, only two phones are detected with weaknesses in this category.

In contrast, the undocumented channels turn out to be highly risky. The mis-exposed system properties account for over 50% (37/65) of the leakages, and another 10 leakages are through the system settings. In fact, most leakages (over 70%) are through these two channels. In particular, all the newly found miscellaneous UUIs are caught through them. This should urge the manufacturers to restrain from using them to share or store device properties and configurations.

The mis-regulation of system services causes another 5 leakages. The functions that can be called through Java reflection, in fact, are an intermediate when serving corresponding documented APIs, as we discussed in Section VI-B. All intermediates in the call trace should be taken into regulation, as AOSP does for most of its APIs. Nonetheless, this principle has not been well followed by OEM OSes on their customized service managers.

> **Findings in RQ2:** *The vast majority of vulnerable UUIs are in the chip and cellular category. The system properties are the major exfiltration points that reserve over 50% of total leakages. The remaining two undocumented*

TABLE VI: List of apps found potential UUI collection

| # | App Package Name | Market & Downloads* | Involved UUIs | Affected Devices | Access Channel |
|---|---|---|---|---|---|
| 1 | com.kwai.m2u | 🔴 81mil+ | #2, Misc.[†] | $\underline{B_9}$,$\underline{C_{10,11}}$,$\underline{D_{10,11}}$ | Properties |
| 2 | com.kwai.videoeditor | 🔴 86mil+ | #2, Misc.[†] | $\underline{B_9}$,$\underline{C_{10,11}}$,$\underline{D_{10,11}}$ | Properties |
| 3 | com.lbe.parallel.intl | ▶ 100mil+ | #4 | $\underline{G_{10}}$ | Services |
| 4 | com.liulishuo.engzo | 🔴 116mil+ | #2 | $\underline{G_{10}}$ | Properties |
| 5 | com.oppo.store | 🔴 3mil+ | Misc.[†] | $\underline{C_{10,11}}$, $\underline{D_{10,11}}$ | Properties |
| 6 | com.renrendai.haohuan | 🔴 25mil+ | #2 | $\underline{G_{10}}$ | Properties |
| 7 | com.tencent.qqimsecure | 🔴 682mil+ | #4 | Unidentified | Settings |
| 8 | com.tencent.wifimanager | 🔴 192mil+ | #4 | Unidentified | Settings |
| 9 | com.wuba.zhuanzhuan | 🔴 234mil+ | #2 | $\underline{G_{10}}$ | Properties |
| 10 | com.xunmeng.merchant | 🔴 50mil+ | #1 | $\underline{H_{10,11}}$ | Properties |
| 11 | com.xunmeng.pinduoduo | 🔴 5bil+/▶ 500k+ | #1 | $\underline{H_{10,11}}$ | Properties |
| 12 | ru.yandex.searchplugin | ▶ 100mil+ | #2 | $\underline{G_{10}}$ | Properties |

\* The apps labelled with 🔴 are downloaded from Xiaomi App Store, with the statistics provided by Kuchuan.com, those with ▶ are downloaded from Google Play Store.
[†] The caught UUI access is through requesting the system property `gsm.serial`.

---

access channels, system services and system settings, account for 5 and 10 leakages respectively. All newly found miscellaneous UUIs are found through system settings and system properties.

### C. RQ3: UUI Collection by Existing Apps

The pervasiveness of UUI mishandling weaknesses on Android devices has provided a hotbed for excessive UUI collection that threatens user privacy. Whether the identified exfiltration points have been (ab)used by existing apps in the wild before we reported to the manufacturers has greatly concerned us. We thus conduct a small-scale study to investigate this. We note that our study only focuses on whether existing apps use the collection methods we have identified in this work, and large-scale studies on apps' data collection behaviors and purposes can be found in [58], [64], [74], [88].

To be representative, we collect apps from both the official Google Play Store and a popular alternative app store called Xiaomi App Store [80]. From each store, we crawl the top 150 apps, which are either mostly installed or increasingly popular in recent months. We exclude those apps appearing in top 150 in Xiaomi App Store while crawling from Google Play Store, and thus we have collected 300 distinct apps for our testing. Nonetheless, for the apps appearing in both stores (overall 16), we keep both of their packages, so that we can investigate whether they behave consistently. All collected apps have at least one million user installations, and they cover various types such as social networking, financial service, and online shopping.

Since the access channels U2-I2 identifies are all Java-level interfaces, our study focuses on only the `.class` files of the collected apps. We reverse engineer them into Smali code, and scan it for the occurrences of API names of the documented interfaces, methods for reading system settings (i.e., getter methods in the `Settings` class) and system properties (i.e., `getProperty()` in the `System` class), and methods of Java reflection and the names of system service managers. We manually confirm the reported suspicions, and our findings are summarized in Table VI. Out of the 300 scanned apps, 12

apps have used the collection methods. Two apps are found to retrieve two UUIs, and all others retrieve one. Only one app (#11 in Table VI) appears in both stores. We analyze both packages of this app and find they have no inconsistent behaviors on UUI collection.

No suspicious UUI collection through the documented interfaces is found. This may owe to the following two reasons.

1) *Developers' awareness*. Due to the clear reminders in AOSP documentation, the app developers may have understood that invoking those documented APIs leads to a security exception when their apps are running on a device installed with the latest Android.
2) *IDE inspection*. The development environment (e.g., Android Studio) can warn the developers for the invocation of APIs that are illegitimate or unrecommended in any version in the app's release targets.

All three undocumented access channels are involved in the reported suspicions, including one through system services, two through the system settings provider, and nine through system properties. The device ID is the most popular UUI, and 6 out of 12 apps attempt to access it. Three apps (#1, #2 and #5 in Table VI) use the key `gsm.serial` to query the system properties. This property is identified as a miscellaneous UUI (see Table IV), and it stores the ID of the PCB chip in phones from *Oppo* ($\underline{D_{10,11}}$) and *OnePlus* ($\underline{C_{10,11}}$). App #5 named `com.oppo.store` is an e-shopping app developed by the parent company of *Oppo* and *OnePlus*, who is apparently aware of this hidden channel. The other two apps (apps #1 and #2) are developed by the same company, but the reason why it is aware of the channel is unknown. We have not figured out how it links to the two manufacturers. Two apps (apps #7 and #8) query the system settings with the key `default_sim1_value`. This key name is similar to those under the IMSI category in Table III, and thus we speculate it to be the IMSI. Nonetheless, which device includes this setting remains unknown (and thus they are labelled as *unidentified* in Table VI). One app (#3 in Table VI) involves the hacking way to invoke the system services. It attempts to call a method `vivoTelephonyApi()` from the class `vivoTelephonyApiParams`, which is implemented by *Vivo* ($\underline{G}$) to access the IMSI.

```
1  // frameworks/base/telephony/common/com/android/internal/
       telephony/TelephonyPermissions.java
2  public static boolean checkReadDeviceIdentifiers(Context
       context, ..., int uid, String callingPackage, ...) {
3   final int appId = UserHandle.getAppId(uid);
4   if(appId == Process.SYSTEM_UID || appId == Process.ROOT_UID)
5    return true;
6   ...
7   if(... || !isPackageNameInDeviceIdWhiteList(callingPackage))
8    return reportAccessDeniedToReadIdentifiers(..., uid,
       callingPackage, ...);
9   return true;
10 }
```

Fig. 5: The implementation of permission checking that raises a whitelisting issue (line 7) on a Huawei device

---

**Findings in RQ3:** *The identified channels for UUI collection have been used by existing popular apps from major app stores. An extensive UUI collection by apps in the wild is not observed though, with only 12 out of 300 apps being found to have relevant behaviors. All three access channels have been used for UUI collection.*

---

## VIII. WHITELISTING ISSUES

We come across a hidden security issue in our experiments on a phone of *Huawei* (A10), which is the first OEM phone we test. We happen to embed several API calls into a sample app, and we find they manage to obtain the serial number and the device ID without any permission of dangerous level or signature level. Surprisingly, after the same snippet of code is copied into our tester app, it fails to obtain any UUI. After exhaustive comparison, we eventually figure out that the sample app could access the UUIs because it happens to be named with the package name of a popular e-commerce app. We dump and investigate the OS image of the device, and we find that it maintains a whitelist for its permission checking (line 7 in Figure 5) in the `TelephonyPermissions` class. The apps listed in the whitelist are allowed to bypass its permission control. Obviously, such whitelisting mechanism is exploitable, as it relies completely on the package name for access control rather than on the app signature. A malicious app could deceive this mechanism by simply naming itself with any package name in the whitelist.

This issue greatly concerns us given its exploitability. We thus investigate it on all phones. We dump their images and analyze the classes for permission checking, such as `TelephonyPermissions` and `WifiPermissionsUtil`. We manage to find three whitelisting issues from the devices of *Huawei*, *Oppo* and *Vivo*. Two are related to UUI access, and one is related to GPS data collection, which is another privacy-sensitive data that has been extensively studied [58], [59], [60], [64], [74]. All of them validate the access through app names only.

- In the devices of *Huawei* (A10), **253** apps are whitelisted for chip and cellular UUIs (UUIs #1-4). They are from various categories such as online shopping, online travel agents, food delivery, and productivity.

- In the devices of *Oppo* (D10,11), **272** apps are whitelisted for background access to GPS location. They include leading apps for transportation, food delivery and ride-hailing, and so on. They are all system apps or apps developed by the manufacturer, such as app store, theme app, launcher app and mobile wallet. All their package names are hardcoded in the OS image.

- In the devices of *Oppo* (D10,11) and *Vivo* (G10), **3** map apps are whitelisted for WiFi connection details (SSID and BSSID). Their package names are also hardcoded in the OS image.

## IX. DISCUSSIONS

In this section, we present our insights on the causes of the UUI mishandling weaknesses and provide our recommendations to the involved parties. We also discuss the limitations of our study and outlook the future efforts that are needed.

### A. Lessons Learned from UUI Leakages

The causes of the identified vulnerabilities can be summarized into the following three types. We suggest the manufacturers pay sufficient attention to them in future to enhance the privacy compliance of their devices.

**Defects in AOSP and OEM OSes**. Defects in AOSP and OEM OSes constitute the immediate cause of the identified 51 vulnerabilities. The vast majority of them occur in OEM OSes, and the root cause is the customized components in these OSes. Defects in AOSP have a wider scope of impact than their counterparts in OEM OSes, as they are likely to be inherited by all OEM devices.

**Inadequate awareness from OEMs**. The identified vulnerabilities are not merely due to unintentional programming errors, but in most cases are consequence of inadequate awareness of privacy protection. Most customizations of the OEMs focus on the functionality but overlook the compliance with Android security and privacy guidelines. Take the device of *Vivo* (G10) as an example. It modifies AOSP's telephony manager and exposes a public method `vivoTelephonyApi()` in a system service named "phone". This method returns IMSI with `READ_PHONE_STATE` permission only, which is at dangerous level and should have been abolished for access to IMSI. As another example, the Bluetooth MAC address should be randomized since Android 6, but the actual MAC address can still be obtained through an API on *Smartisan*'s device installed with Android 10 that is released four years later than Android 6.

**Opened channels in OEM OSes**. It is shown that some manufacturers whitelist third-party apps to facilitate their UUI access (see Section VIII), while Google keeps making efforts to tighten the UUI access regulation in the latest versions. We search their documentations for the whitelisting mechanism and find that *Huawei* discloses the apps in its whitelist. It declares that the purpose of the whitelist is to ease the burden of their users on manually granting dangerous-level permissions to those popular apps. In addition, the apps developed by *OnePlus* and *Oppo* (app #5 in Table VI)) use the hidden channels on the devices manufactured by themselves.

12

### B. Recommendations to Involved Parties

We present a few recommendations and mitigation strategies for manufacturers and authorities to suppress UUI leakage.

**Strengthening permission enforcement**. Manufacturers are advised to keep in sync with the security and privacy guidelines of AOSP. For example, the permission READ_PRIVILEDGED_PHONE_STATE has replaced the READ_PHONE_STATE permission in reading some UUIs, but we find 12 weaknesses caused by failing to comply with this change. Manufacturers should perform a thorough checking of their implementation against AOSP's guidelines, especially in revised or customized components.

**Avoiding sharing through system settings/properties**. Sharing resources through the system settings and the system properties are efficient and straightforward in certain scenarios. However, some of these places, such as system settings, are not designated to store UUIs. Manufacturers should restrain from sharing UUIs through them. In addition, they are advised to deploy a rigorous access control in the system settings provider and to enforce rules through SELinux to regulate the access to the system properties.

**Rigorous permission control**. Manufacturers are advised to avoid deploying/leaving opened channels (e.g., the whitelist) for UUI access because this violates the integrity of the permission control, and infringes the user's *right to be informed* and *right to consent*. In case of some system apps that are necessary to be granted with privileged permissions, they should strictly follow Google's official guideline [30] and enforce appropriate access control. Manufacturers should also be aware that the handles of the non-SDK interfaces [31] may be obtained using reflection and should deploy consistent restrictions over them.

**Research community & authority efforts**. The existing studies on Android data collection have been focusing on regulating the behaviors of third-party apps. To complete the privacy protection in the entire ecosystem, we advocate that efforts need to be put into OS-level protection. Authorities should put in place the regulations on the UUI creation, storage and processing of devices. Our study should also motivate privacy checking in other domains [47], [49], [81].

### C. Limitations

To the best of our knowledge, this work is the first one focusing on the OS-level UUI mishandling. However, it carries several limitations that we target to address in future work.

**Scope of UUIs**. Besides UUIs, some near-persistent IDs and non-ID information can also be abused for user tracking or profiling, as revealed in existing studies [5], [7], [11], [68], e.g., installed packages and location data. In this paper, we only focus on those unarguably unresettable IDs to avoid ambiguity.

**Access channels**. U2-I2 explores undocumented access channels centered around six types of UUIs that are identified from Android documentation and literature. It is possible that it covers only part of all UUI access channels. The manufacturer may introduce other hidden ones. Besides that, our study analyzes UUI leakage in the Java context only. It may miss access channels, if any, that leak a UUI in the native context of Android.

**Missed leakages**. Our assessment of system service assumes the UUI access is through a public getter function, like most APIs do. It may result in false negative when an interface leaks UUI through a private function and/or in a more complicated manner.

**Scale of our assessment**. Our tested phones cover those popular models from major manufacturers. We presume the top-selling models of a manufacturer can represent all devices produced by it. Due to device availability, we skip the devices of the same model but for different markets around the world. Moreover, our study on the existing apps reveals some facts but remains limited on its scale compared with the studies on Android data harvesting [58], [64], [74], [88]. We take the large-scale investigation of the UUI collection by third-party apps as future work.

Moreover, our study does not cover Android 12 (API level 31) as it has not been widely adopted by vendors at the moment of this paper being drafted. However, we progressively run our testing on new models we can find. We have found some relevant vulnerabilities on Android 12 devices, reported them, and received a few acknowledgments and CVE registrations, e.g., CVE-2022-30753, CVE-2022-27822, and CVE-2022-22272. All these three CVEs are related to the leakage of vendor-customized UUIs through undocumented channels.

## X. RELATED WORK

Threat hunting has recently emerged as a popular topic in the cybersecurity domain. It describes a solution to detect the potential flaws or malicious activities at the earliest possible time. This section briefs relevant research on Android that fall into this topic.

**Access control**. The access control mechanism of Android is a popular area in the security research community. There are several analysis tools [4], [18], [37] available to scan for permission leakages since the early Android versions. [19] studies the permission re-delegation issue though inter-process and inter-application communications. Later research extends the analysis with broader coverage such as the authorization logic within Android middleware [34], system services [35] and filesystem [36]. Scanning inconsistencies in security policy enforcement is another direction to assess the Android access control system [1], [63]. One recent work IAceFinder [89] finds over twenty inconsistencies in the latest Android OSes through the analysis across the Java classes and native context.

**App-level data gathering**. Protecting sensitive user data against malicious apps has been a persistent topic ever since the birth of Android. Various types of data that third-party apps can access through Android APIs, such as the device IDs [14], [20], [60], geographic locations [68], WiFi status [12], [52], motion sensor data [10], speakers' characteristics [92], and even personalized configurations [44], [53], [56], have been shown to facilitate privacy leakage. Many efforts have also been put into their protection [38], [39], [40], [71], [82].

Recent studies have been mostly focusing on the behaviors of third-party apps in gathering *personally identifiable information* (PII). Gibler *et al.* [20] uses static taint analysis to identify relevant API invocations in third-party apps. Papadopoulos *et al.* [54] summarizes 32 types of PII and conducts a network traffic analysis to understand app-level data collection. Later

studies analyze apps' PII collection through either network traffic [57], [59], [60], [68] or logs files [77]. The feasibility of persistent identifiers harvesting by unprivileged apps is demonstrated in [58]. Shen *et al.* [64] recently presented a large-scale study on the landscape of app-level data collection, including the types and destination of the collected data. In our work, we refer to these studies for UUI recognition, focusing on the complementary problem of UUI handling by the OSes.

**Sensitive data access on Android**. Many studies have been conducted to identify channels of circumventing the permission system of Android. The process information file system (`procfs`) used to be a popular exfiltration point. Zhang *et al.* [85] infer key-logging activities through exploitation over `procfs`. Besides that, studies [6], [13] figure out that the app's activity could also be inferred through the interpretation of data within `procfs`. [42] learns a secret from process footprint to find out what web pages the user is browsing and consequently extracts more fine-grained information about the user's privacy. Zhou *et al.* [90] also take advantage of `procfs` for precise user identification. Spreitzer *et al.* [65] proposes a tool called *ProcHarvester* that boosts `procfs` exploitation as a mature and automatic technique. However, Google has fixed this exfiltration point by gradually tightening its access since Android 7.

Other research has also discovered various methods to access sensitive data on Android. Different covert/side channels, such as the power channel and the browser cache, have been revealed by them [50], [58], [66]. Wang *et al.* [74] study cross-library data harvesting to stealthily read and upload user data. Additional channels that allow apps to access private data include system settings and running services, which are adopted by [6] and [69]. Our work evaluates existing approaches and then proposes three categories of access channels that remain accessible in the latest Android.

## XI. Conclusion

In this work, we present the first comprehensive study of UUI protection in Android phones to complement the existing studies of app-level data harvesting. We design and implement U2-I2, which uses a set of analysis techniques to discover and assess UUI access channels. The access channels are automatically tested by U2-I2 to identify weaknesses that open any attacking opportunity. We have evaluated the most popular 13 phones from 9 manufacturers, which are installed with the official AOSP and OEM OSes, and identified 51 unique vulnerabilities from them. Our work reveals that the UUI mishandling is pervasive in current Android phones, caused by errors in documented APIs, mismanagement of system properties and system settings, and misregulation of system services. Our study should raise an alert to the device manufacturers and policymakers, especially in an era where strict personal data protection laws and regulations are put in place around the world.

## References

[1] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "Acedroid: Normalizing diverse android access control checks for inconsistency detection." in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[2] A. Arora, S. Garg, and S. K. Peddoju, "Malware detection using network traffic analysis in android based mobile devices," in *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*. IEEE, 2014, pp. 66–71.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012, pp. 217–228.

[5] R. Binns, U. Lyngs, M. Van Kleek, J. Zhao, T. Libert, and N. Shadbolt, "Third party tracking in the mobile ecosystem," in *Proceedings of the 10th ACM Conference on Web Science*, 2018, pp. 23–31.

[6] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it:UI state inference and novel android attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[7] T. Chen, I. Ullah, M. A. Kaafar, and R. Boreli, "Information leakage through mobile analytics services," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014, pp. 1–6.

[8] G. Cobucci, "Xiaomi: data protection and privacy at the highest levels thanks to the ISO / IEC 27701 certification," 2020, (accessed 13 April 2022). [Online]. Available: https://en.xiaomitoday.it/xiaomi-e-sicura-protezione-dati-privacy.html

[9] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Analyzing android encrypted network traffic to identify user actions," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 1, pp. 114–125, 2015.

[10] A. Das, N. Borisov, and M. Caesar, "Tracking mobile web users through motion sensors: Attacks and defenses." in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[11] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android." in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[12] A. Di Luzio, A. Mei, and J. Stefa, "Mind your probes: De-anonymization of large crowds through smartphone wifi probe requests," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[13] W. Diao, X. Liu, Z. Li, and K. Zhang, "No pardon for the interruption: New inference attacks on android through interrupt timing analysis," in *IEEE Symposium on Security and Privacy*, 2016, pp. 414–432.

[14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[15] European Comission, "Directive 2009/136/EC of the European Parliament and of the Council," 2009, (accessed 13 April 2022). [Online]. Available: https://edps.europa.eu/sites/default/files/publication/dir_2009_136_en.pdf

[16] ——, "Data protection in the EU," 2021, (accessed 13 April 2022). [Online]. Available: https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu

[17] Fedral Trade Commission, "A Brief Overview of the Federal Trade Commission's Investigative, Law Enforcement, and Rulemaking Authority," 2019, (accessed 13 April 2022). [Online]. Available: https://www.ftc.gov/about-ftc/what-we-do/enforcement-authority

[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 627–638.

[19] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[20] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[21] Google, "User Data - Play Console Help," 2020, (accessed 13 April 2022). [Online]. Available: https://support.google.acm/googleplay/android-developer/answer/10144311?hl=en&ref_topic=9877467

[22] ——, "Add System Properties," 2021, (accessed 2 May 2022). [Online]. Available: https://source.android.com/devices/architecture/configuration/add-system-properties

[23] ——, "Android Code Search," 2021, (accessed 13 April 2022). [Online]. Available: https://cs.android.com/android/platform/superproject/+/master:frameworks/base/telephony/java/android/telephony/SubscriptionInfo.java

[24] ——, "Android Security Bulletin—April 2021," 2021, (accessed 13 April 2022). [Online]. Available: https://source.android.com/security/bulletin/2021-04-01

[25] ——, "Best practices for unique identifiers," 2021, (accessed 13 April 2022). [Online]. Available: https://developer.android.com/training/articles/user-data-ids

[26] ——, "Device Identifiers," 2021, (accessed 13 April 2022). [Online]. Available: https://source.android.com/devices/tech/config/device-identifiers

[27] ——, "Privacy changes in Android 10," 2021, (accessed 13 April 2022). [Online]. Available: https://developer.android.com/about/versions/10/privacy/changes

[28] ——, "Privacy in Android 11," 2021, (accessed 13 April 2022). [Online]. Available: https://developer.android.com/about/versions/11/privacy

[29] ——, "Privacy: MAC Randomization," 2021, (accessed 13 April 2022). [Online]. Available: https://source.android.com/devices/tech/connect/wifi-mac-randomization

[30] ——, "Privileged Permission Allowlisting," 2021, (accessed 13 April 2022). [Online]. Available: https://source.android.com/devices/tech/config/perms-allowlist

[31] ——, "Restrictions on non-SDK interfaces," 2021, (accessed 13 April 2022). [Online]. Available: https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces

[32] ——, "Android Interface Definition Language (AIDL)," 2022, (accessed 18 August 2022). [Online]. Available: https://developer.android.com/guide/components/aidl

[33] ——, "<permission>," 2022, (accessed 26 March 2022). [Online]. Available: https://developer.android.com/reference/android/os/IInterface

[34] S. A. Gorski, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, and A. Bartel, "Acminer: Extraction and analysis of authorization checks in android's middleware," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, 2019, pp. 25–36.

[35] S. A. Gorski III and W. Enck, "Arf: identifying re-delegation vulnerabilities in android system services," in *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*, 2019, pp. 151–161.

[36] S. A. Gorski III, S. Thorn, W. Enck, and H. Chen, "Fred: Identifying file re-delegation in android system services," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[37] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *The Network and Distributed System Security Symposium (NDSS)*, 2012.

[38] M. Gruteser and D. Grunwald, "Enhancing location privacy in wireless lan through disposable interface identifiers: a quantitative analysis," *Mobile Networks and Applications*, vol. 10, no. 3, pp. 315–325, 2005.

[39] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A programmable interface for extending android security," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[40] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[41] N. Hua, Y. Qiang, W. Yanhong, W. Teng, J. Ke, Y. Yinan, L. Cheng, H. Ying, D. Yun, Z. Fei, D. Youjun, Y. Quan, and W. Haoqian, "Mobile intelligent terminal and application software user personal information protection implementation guide, Part 1:General Principle," Telecommunication Terminal Industry Forum Association (China), Tech. Rep. T/TAF 070-2020, September 2020, (accessed 13 April 2022). [Online]. Available: http://www.taf.net.cn/StdDetail.aspx?uid=6C9F4148-84AA-4990-A3F8-6B11AD13D237&stdType=TAF

[42] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *IEEE Symposium on Security and Privacy*, 2012.

[43] A. Khani, "Xiaomi Reacted to the Article regarding its Privacy Policy by FORBES," 2020, (accessed 13 April 2022). [Online]. Available: https://gadgettrait.com/xiaomi-reacted-to-the-article-regarding-its-privacy-policy-by-forbes/

[44] A. Kurtz, H. Gascon, T. Becker, K. Rieck, and F. C. Freiling, "Fingerprinting mobile devices using personalized configurations." *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 1, pp. 4–19, 2016.

[45] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner, "Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[46] C. Leung, J. Ren, D. Choffnes, and C. Wilson, "Should you use the app for that? comparing the privacy implications of app-and web-based online services," in *Proceedings of the 2016 Internet Measurement Conference*, 2016, pp. 365–372.

[47] Y. Ling, K. Wang, G. Bai, H. Wang, and J. S. Dong, "Are they toeing the line? diagnosing privacy compliance violations among browser extensions," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[48] N. Lomas, "Google's Android ad ID targeted in strategic GDPR tracking complaint," 2020, (accessed 13 April 2022). [Online]. Available: https://techcrunch.com/2020/05/13/googles-android-ad-id-targeted-in-strategic-gdpr-tracking-complaint

[49] K. Mahadewa, Y. Zhang, G. Bai, L. Bu, Z. Zuo, D. Fernando, Z. Liang, and J. S. Dong, "Identifying privacy weaknesses from multi-party trigger-action integration platforms," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 2–15.

[50] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 51–60.

[51] S. Nath, "Madscope: Characterizing mobile in-app targeted ads," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 59–73.

[52] L. Nguyen, Y. Tian, S. Cho, W. Kwak, S. Parab, Y. Kim, P. Tague, and J. Zhang, "Unlocin: Unauthorized location inference on smartphones without being caught," in *2013 International Conference on Privacy and Security in Mobile Systems (PRISMS)*, 2013.

[53] G. Palfinger and B. Prünster, "Androprint: analysing the fingerprintability of the android api," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[54] E. P. Papadopoulos, M. Diamantaris, P. Papadopoulos, T. Petsas, S. Ioannidis, and E. P. Markatos, "The long-standing privacy debate: Mobile websites vs mobile apps," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 153–162.

[55] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 71–72.

[56] L. Qiu, Z. Zhang, Z. Shen, and G. Sun, "Apptrace: Dynamic trace on android devices," in *2015 IEEE International Conference on Communications (ICC)*, 2015, pp. 7145–7150.

[57] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan,

M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[58] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[59] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "A longitudinal study of pii leaks across android app versions," in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[60] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes, "Recon: Revealing and controlling pii leaks in mobile network traffic," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 361–374.

[61] Samsung, "Samsung Galaxy Smartphones and Tablets Join Android Enterprise Recommended Program," 2020, (accessed 13 April 2022). [Online]. Available: https://news.samsung.com/global/samsung-galaxy-smartphones-and-tablets-join-android-enterprise-recommended-program

[62] ——, "Knox certificates and guidance," 2021, (accessed 13 April 2022). [Online]. Available: https://www.samsungknox.com/en/knox-platform/knox-certifications

[63] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework." in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[64] Y. Shen, P.-A. Vervier, and G. Stringhini, "Understanding worldwide private information collection on android," in *The Network and Distributed System Security Symposium (NDSS)*, 2021.

[65] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard, "Procharvester: Fully automated analysis of procfs side-channel leaks on android," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 749–763.

[66] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[67] StatCounter, "Mobile Vendor Market Share Worldwide, Apr 2021 - Apr 2022," 2022, (accessed 6 May 2022). [Online]. Available: https://gs.statcounter.com/vendor-market-share/mobile/worldwide

[68] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Workshop on Mobile Security Technologies (MoST)*, vol. 10. Citeseer, 2012.

[69] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.

[70] The European Parliament, "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, and repealing directive 95/46/ec (general data protection regulation) (text with eea relevance)," *Official Journal of the European Union*, 2016.

[71] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.-W. Chen, "Turtle guard: Helping android users apply contextual privacy preferences," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, 2017, pp. 145–162.

[72] UNCTAD, "Data Protection and Privacy Legislation Worldwide," 2020, (accessed 13 April 2022). [Online]. Available: https://unctad.org/page/data-protection-and-privacy-legislation-worldwide

[73] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[74] J. Wang, Y. Xiao, X. Wang, Y. Nan, L. Xing, X. Liao, J. Dong, N. Serrano, H. Lu, X. Wang *et al.*, "Understanding malicious cross-library data harvesting on android," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[75] K. Wang, J. Zhang, G. Bai, R. Ko, and J. S. Dong, "It's not just the site, it's the contents: Intra-domain fingerprinting social media websites through cdn bursts," in *Proceedings of the Web Conference (WWW)*, 2021, pp. 2142–2153.

[76] P. Wang, K. Liu, L. Jiang, X. Li, and Y. Fu, "Incremental mobile user profiling: Reinforcement learning with spatial knowledge graph for modeling event streams," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 853–861.

[77] Z. Wang, Z. Li, M. Xue, and G. Tyson, "Exploring the eastern frontier: A first look at mobile app tracking in china," in *International Conference on Passive and Active Network Measurement*. Springer, 2020, pp. 314–328.

[78] B. Wolford, "What are the GDPR fines?" 2022, (accessed 6 May 2022). [Online]. Available: https://gdpr.eu/fines/

[79] Q. Xiao, M. K. Reiter, and Y. Zhang, "Mitigating storage side channels using statistical privacy mechanisms," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1582–1594.

[80] Xiaomi, "Top List - Xiaomi App Store," 2021, (accessed 13 April 2022). [Online]. Available: https://app.mi.com/topList

[81] F. Xie, Y. Zhang, C. Yan, S. Li, L. Bu, K. Chen, Z. Huang, and G. Bai, "Scrutinizing privacy policy compliance of virtual personal assistant apps," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[82] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[83] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1043–1054.

[84] Z. Yang and C. Yue, "A comparative measurement study of web tracking on mobile and desktop environments," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 24–44, 2020.

[85] K. Zhang and X. Wang, "Peeping tom in the neighborhood: keystroke eavesdropping on multi-user systems," in *18th USENIX security symposium (USENIX Security 2009)*, 2009.

[86] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on android," in *IEEE Symposium on Security and Privacy*, 2015, pp. 915–930.

[87] X. Zhang, X. Wang, R. Slavin, T. Breaux, and J. Niu, "How does misconfiguration of analytic services compromise mobile privacy?" in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 1572–1583.

[88] S. Zhao, G. Pan, Y. Zhao, J. Tao, J. Chen, S. Li, and Z. Wu, "Mining user attributes using large-scale app lists of smartphones," *IEEE Systems Journal*, vol. 11, no. 1, pp. 315–323, 2016.

[89] H. Zhou, H. Wang, X. Luo, T. Chen, Y. Zhou, and T. Wang, "Uncovering cross-context inconsistent access control enforcement in android," in *The Network and Distributed System Security Symposium (NDSS)*, 2022.

[90] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 1017–1028.

[91] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *IEEE Symposium on Security and Privacy*, 2014, pp. 409–423.

[92] Z. Zhou, W. Diao, X. Liu, and K. Zhang, "Acoustic fingerprinting revisited: Generate stable device id stealthily with inaudible sound," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 429–440.

# APPENDIX

## A. UUI Access of Two Example APIs

**getImei().** The definition of this API is shown in the code snippet below (lines 2-4). It calls another method within the same class (lines 6-14). As shown in line 8, the IMEI is

obtained from calling the method getImeiForSlot() of an object of ITelephony.

```
1   /* .../android/telephony/TelephonyManager.java */
2   public String getImei() {
3    return getImei(getDefaultSim());
4   }
5   ...
6   @Deprecated
7   public String getImei(int slotIndex) {
8    ITelephony telephony = getITelephony();
9    ...
10   try {
11    return telephony.getImeiForSlot(...);
12   }
13   ...
14  }
```

The ITelephony is a nested interface class used to interact with the system service phone, in which the getImeiForSlot() interface is defined in an inner class named Proxy (lines 6-10, in the code snippet of ITelephony class). The proxy is generated by another inner class of ITelephony called Stub (lines 2-13). The remote interface is defined in the stub class (line 3) and shares the same method name with the local interface. Both the stub and proxy are inner classes *generated* at compilation time based on the predefined Android Interface Definition Language (AIDL) file to allow apps binding [32]. The code snippet including both the local and remote interfaces is shown below.

```
1   /* .../com/android/internal/telephony/ITelephony.java */
2   public static abstract class Stub extends android.os.Binder
          implements com.android.internal.telephony.ITelephony {
3    public java.lang.String getImeiForSlot(...) throws android.os
          .RemoteException;
4    ...
5    private static class Proxy implements com.android.internal.
          telephony.ITelephony {
6     @Override public java.lang.String getImeiForSlot(...)
          throws android.os.RemoteException {
7      ...
8      boolean _status = mRemote.transact(Stub.
          TRANSACTION_getImeiForSlot, _data, _reply, 0);
9      ...
10    }
11   }
12  }
```

The remote interface is overridden in the class PhoneInterfaceManager, shown in the code snippet below. The IMEI is returned by calling phone.getImei() (line 7).

```
1   /* .../package/services/Telephony/src/com/android/phone/
          PhoneInterfaceManager.java */
2   @Override
3   public String getImeiForSlot(...) {
4    Phone phone = PhoneFactory.getPhone(slotIndex);
5    ...
6    try {
7     return phone.getImei();
8    }
9    ...
10  }
```

The Phone appearing in the above code snippet is an abstract class. Its method getImei() is overridden by two classes, ImsPhoneBase and GsmCdmaPhone. The former class is invoked when the device has no IMEI (lines 1-5 of the code

snippet below), and the latter corresponds to the case where the device has a valid IMEI (lines 6-10).

```
1   /* .../com/android/internal/telephony/imsphone/ImsPhoneBase.
          java */
2   @Override
3   public String getImei() {
4    return null;
5   }
6   /* .../com/android/internal/telephony/GsmCdmaPhone.java */
7   @Override
8   public String getImei() {
9    return mImei;
10  }
```

**getSerial()**. This API is provided in the class Build, as shown in the code below. It creates an object of the stub class of IDeviceIdentifiersPolicyService (line 3) and invokes the local interface (i.e., getSerialForPakcage() in line 6).

```
1   /* .../android/os/Build.java */
2   public static String getSerial() {
3    IDeviceIdentifiersPolicyService service =
          IDeviceIdentifiersPolicyService.Stub.asInterface(...));
4    try {
5     ...
6     return service.getSerialForPackage(callingPackage, null);
7    }
8    ...
9   }
```

The API's invocation is requested through the local interface defined in the proxy class (lines 5-10 in the code snippet below), and then handled via the remote interface defined in the stub class (lines 2-13). Both the stub and proxy are inner classes of IDeviceIdentifierPolicyService, in which a few interfaces are defined for apps to interact with the system service called device_identifiers.

```
1   /* .../android/os/IDeviceIdentifiersPolicyService.java */
2   public static abstract class Stub extends android.os.Binder
          implements android.os.IDeviceIdentifiersPolicyService
3   {
4    private static class Proxy implements android.os.
          IDeviceIdentifiersPolicyService {
5     @Override public java.lang.String getSerialForPackage(...)
          throws android.os.RemoteException
6     {
7      ...
8      boolean _status = mRemote.transact(Stub.
          TRANSACTION_getSerialForPackage, _data, _reply, 0);
9      ...
10    }
11    ...
12    public java.lang.String getSerialForPackage(...) throws
          android.os.RemoteException;
13   }
```

Next, the remote interface is overridden in DeviceIdentifiersPolicyService. The serial number is returned by calling a getter function of the class SystemProperties (line 5).

```
1   /* .../android/server/os/DeviceIdentidiersPolicyService.java
          */
2   @Override
3   public @Nullable String getSerialForPackage(...) throws
          RemoteException {
4    ...
5    return SystemProperties.get("ro.serialno", Build.UNKNOWN);
6   }
```

Unlike `getImei()` that the IMEI is returned within the scope of a system service, `getSerial()` is finally served by querying system properties. For this reason, the system properties are treated as an undocumented access channel.

### B. Alternative Approaches for Entry Point Retrieval

**System Settings**. Android provides APIs to read data from the settings database but only when the caller knows the key. Therefore, we must find an alternative approach to access system settings without keys. In addition to the ADB approach, we can also retrieve all the keys by running a URI scheme query in an app, as shown in the code below:

```
1  /* Change the "system" to global and secure to query other two
        setting tables */
2  Uri uri = Uri.parse("content://settings/system");
3  Cursor cursor = getContentResolver().query(uri, null, null,
        null, null);
4  while (cursor.moveToNext()) {
5   /* Three strings to be read from the cursor.
6    They are "id", "name" and "values" respectively */
7   Log.d(TAG, cursor.getString(0) + ",␣" + cursor.getString(1) +
        ",␣" + cursor.getString(2));
8  }
```

**System Properties**. The list of all system properties can also be obtained through executing a specific command through the `Runtime` instance, as shown in the snippet below:

```
1  String propertyValue = ""; // Leave it empty to retrieve all
2  try {
3   Process getPropProcess = Runtime.getRuntime().exec("getprop␣"
        + propertyName);
4   BufferedReader bufferReader = new BufferedReader(new
        InputStreamReader(getPropProcess.getInputStream()));
5   propertyValue = bufferReader.readLine();
6   bufferReader.close();
7  } catch (Exception e) { ... }
```

**System Services**. Android does not offer an API to retrieve all system services. However, the list of system services can be retrieved through Java reflection. The code sample is shown below:

```
results = (String[])class.forName("android.os.ServiceManager").
    getMethod("ListServices").invoke(null);
```

Moreover, the list of system services can also be obtained by executing a specific command via the `Runtime` instance, in a similar way to the retrieval of system properties, as shown below:

```
BufferedReader bufferReader = new BufferedReader(new
    InputStreamReader(Runtime.getRuntime().exec("service␣list").
    getInputStream()));
```

### C. Testing System Service

First, U2-I2 retrieves all active system services by the approaches discussed in Section VI-B1 and Appendix B). Given a list of system services, including the service names and the corresponding manager classes, U2-I2 takes advantage of Java reflection to obtain a binder object of each of them. It attempts to call all public methods defined in it. This process is briefly presented in the code snippet below. It scans all services (lines 2-12), and for each service, it enumerates all

public methods to search for any potential UUI leakage from the data returned (lines 22-32).

```
1  public static void testSystemServices(String[] nameList,
        String[] classList) {
2   for (int i=0, i<nameList.length; i++) {
3    Class class = Class.forName(classList[i] + "$Stub");
4    Field[] fields = class.getDeclaredFields();
5    for (int j=1; j<=fields.length; j++) {
6     try {
7      /* Below we invoke an inhouse method of U2-I2 */
8      String result = callMethod(nameList[i], j)
9     }
10    ...
11   }
12  }
13 }
14 public static String callMethod(String serviceName, int code){
15  String result = "";
16  IBinder iBinder = (IBinder) Class.forName("android.os.
        ServiceManager").getMethod("getService",String.class).
        invoke(null, nameList[i]);
17  Parcel data = Parcel.obtain();
18  Parcel reply = Parcel.obtain();
19  /* Below we invoke an inhouse method to get the package name
        of the service class */
20  data.writeInterfaceToken(getInterfaceDescriptor(serviceName))
        ;
21  boolean toRepeat = true;
22  while (toRepeat) {
23   /* Below we invoke an inhouse method to feed arguments into
        data parcel */
24   data = feedArguments();
25   try {
26    iBinder.transact(code, data, reply, 0);
27    result = reply.readString();
28    ...
29   }
30   ...
31   /* Set toRepeat to false if we have obtained a valid output
        or run out of all argument candidates */
32  }
33  return result;
34 }
```

Invoking an unknown method requests a `code` (line 26). It can be exhaustively enumerated from 0 to the maximum allowed value ($2^{24}$-1). To be more efficient, U2-I2 narrows down its range in advance. All `codes` of a service are defined in the interface class as constants of the type `static final int` and are assigned with a unique integer in ascending order. Therefore, a rough range can be estimated by querying the total number of variables/constants declared in that class. This can be achieved by calling `getDeclaredFields()` (line 4).

U2-I2 also needs to set parameters when calling a method (line 24). Since most methods for UUI access in system services are created as *getter* methods, they usually take either no argument or a limited number of primitive and non-primitive types of parameters such as integers indicate the SIM card slot index, or strings represent the pacakage name of the caller. Therefore, U2-I2 sets parameters from a predefined set. This strategy is shown effective in the testing.