On the (In)Security of Non-resettable Device Identifiers in Custom Android Systems

Zikan Dong*[1], Liu Wang*[1], Guoai Xu[2], and Haoyu Wang†[1]
[1]Huazhong University of Science and Technology, China [2]Harbin Institute of Technology, Shenzhen, China
[1]{zikandong,haoyuwang}@hust.edu.cn, [2]xga@hit.edu.cn

***Abstract*—User tracking is critical in the mobile ecosystem and relies on device identifiers to build user profiles. Early versions of Android allowed third-party apps to easily access non-resettable identifiers such as serial numbers and IMEI. As privacy concerns grew, Google has tightened identifier access in native Android. In response, stakeholders in custom Android systems introduced covert channels (e.g., system properties and settings) to maintain consistent and stable identifier access across systems and devices, which undoubtedly increases privacy risks. This paper examines the introduction of such channels through system customization and their vulnerability due to poor access control. We present IDRADAR, a scalable and accurate approach for identifying vulnerable properties and settings in custom Android systems. Applying our approach to 1,814 custom ROMs, we identified 8,192 system properties and 3,620 settings that store non-resettable device identifiers. Among these, 3,477 properties and 1,336 settings lack adequate access control and could be exploited by third-party apps to track users without permissions. Further validation on real devices demonstrates the effectiveness of our approach. Compared to state-of-the-art, IDRADAR offers improved scalability and analytical capabilities. Additionally, we investigate the root causes of the access control deficiencies and observe that such vulnerabilities frequently recur across devices from the same OEMs. We have reported our findings to the respective vendors and received positive confirmations. Our work underscores the need for greater scrutiny of covert access to device identifiers and better solutions to safeguard user privacy during system customizations.**

***Index Terms*—device identifiers, Android security, privacy**

I. Introduction

User tracking is pivotal in the Android ecosystem, enabling personalized services, targeted advertising, and behavior analytics [1]–[4]. To support user tracking, Android relies on identifiers, which are unique strings or numbers used to distinguish users or devices and are often collected alongside other personal data. Their uniqueness allows apps to link user behavior across time, locations, and apps, facilitating the construction of detailed user profiles. Non-resettable device identifiers, such as the International Mobile Equipment Identity (IMEI), are among the most commonly used and capable types of identifiers. They are permanently bound to devices and persist even after a factory reset, making them particularly valuable for continuous tracking. These identifiers are widely used not only by third-party apps but also by various supply chain actors, including OS developers, hardware vendors, and pre-installed apps [5]–[11]. However, these features also pose serious privacy risks. If compromised, non-resettable identifiers can lead to severe breaches of user privacy, potentially exposing extensive personal data to malicious actors. Such risks highlight the critical need for robust protection to safeguard non-resettable identifiers.

Accordingly, Android has introduced significant updates in the management of these identifiers to strengthen user privacy. Prior to Android 10, apps could access these identifiers with the appropriate permissions and user consent. Since Android 10, such access has been restricted to apps with privileged permissions, which are unavailable to third-party apps. Google now recommends that third-party apps use resettable advertising IDs for analytics and advertising. Meanwhile, privacy regulations such as the GDPR [12] and CCPA [13] have further tightened controls on the collection and use of personal data, including device identifiers [14].

In response to such restrictions, some supply chain actors introduce *covert channels* through system customization to facilitate access to non-resettable identifiers [15,16]. A common practice is to store these identifiers in system properties and settings, which act as global variables and can be easily accessed across the systems. This practice aims to maintain consistent and convenient tracking across different system versions or customizations, thereby bypassing restrictions imposed by Google or Original Equipment Manufacturers (OEMs). Such covert access channels, however, if not properly secured, can leak stable device identifiers and create new attack surfaces. They must therefore be protected by access control policies as strict as the identifiers themselves. Unfortunately, vendor customizations often introduce misconfigurations and outdated security mechanisms, as demonstrated in prior studies [17]–[22].

The security of custom Android systems has been widely studied by the research community. Prior studies have mainly focused on privacy risks in pre-installed apps [6,19,23], and the effectiveness of security updates in custom environments [18, 20,21]. While these studies have touched on privacy concerns in custom systems, they primarily consider official tracking channels and overlook covert channels introduced through customization. To the best of our knowledge, the only study that examines such covert channels is U2-I2 [15], which relies on a dynamic testing method. It leverages a fundamental characteristic of non-resettable identifiers that they remain stable across factory resets but differ across devices. Building on this, U2-I2 repeatedly collects values of system properties

* Zikan Dong and Liu Wang contributed equally to this research.
† Haoyu Wang (haoyuwang@hust.edu.cn) is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00152

and settings from the same device (before and after factory resets) and from multiple devices of the same model. Entries that remain constant on a device yet vary across devices are flagged as potential identifiers and then manually verified. While effective, this approach faces two major limitations. First, it is not scalable: it requires repeated factory resets, direct physical device access, and labor-intensive manual testing. Second, dynamic testing inherently provides limited coverage, since it can only capture entries activated during execution, leaving unused ones unobserved and potentially missing vulnerabilities. Consequently, the study was confined to 13 smartphones from 9 vendors and revealed 30 vulnerabilities.

In this work, we aim to conduct a large-scale, extensible study on covert channels introduced through system customization. To this end, we present IDRADAR, an end-to-end approach designed to identify sensitive and vulnerable system properties and settings that store non-resettable device identifiers in custom Android systems. Unlike U2-I2, IDRADAR adopts a static analysis approach that relies solely on system ROMs[1] without requiring physical devices, enabling scalable analysis across both legacy and recent versions as well as diverse vendor-branded ROMs. Moreover, static analysis provides system-wide coverage, allowing for a more comprehensive examination than dynamic testing. These two advantages enable IDRADAR to overcome the limitations of U2-I2 and achieve broader coverage. Specifically, our approach operates in three stages. First, we locate the usage of system properties and settings in custom ROMs. This process begins with a pilot study to catalog the various methods used to access these properties and settings, followed by static analysis leveraging control- and data-flow techniques to extract usage points across the codebase. Second, we identify candidate properties and settings that may store non-resettable device identifiers using two heuristic rules, and then manually analyze the code context to verify their content, pinpointing sensitive system properties and settings that store such identifiers. Finally, by examining access control policies in custom systems, we uncover vulnerable properties and settings that expose non-resettable identifiers without adequate protection, i.e., those that can be abused by third-party apps without permission.

We applied our approach to 1,814 custom Android ROMs from 250 OEMs and identified 8,192 system properties and 3,620 system settings containing non-resettable device identifiers. Among them, 3,477 (42.4%) properties and 1,336 (36.9%) settings across 1,112 (61.3%) systems lacked proper access control. Our analysis attributes these vulnerabilities to three main causes: (*i*) overlooked properties and settings, (*ii*) overly complex access control rules, and (*iii*) covert supply chain actor involvement. To validate our findings, we developed a test app that dynamically retrieves all system properties and settings on a device. On 32 real devices, our approach identified 130 vulnerable entries. Among them, 102 were confirmed using our test app, and the remaining 28 required specific conditions, validating our detection results. We also compared our method with U2-I2 [15] on four physical devices. The results show that our approach can detect more identifiers, especially those tied to conditional triggers, demonstrating its advantage in conducting comprehensive system-wide analysis. Additionally, we found that vulnerable implementations often recur across devices from the same OEMs, likely due to reused code or shared configuration practices. We reported all identified issues to the corresponding vendors, and 84 vulnerabilities have been confirmed.

In summary, our work presents these contributions:

- We conduct a large-scale study on custom Android systems to examine covert channels (i.e., custom system properties and settings) used to access non-resettable identifiers. Our study presents an overall landscape of the privacy issues introduced by these channels.
- We present IDRADAR, a systematic analysis framework for investigating covert channels in custom Android systems. It identifies the system properties and settings that expose non-resettable identifiers and detects access control weaknesses that could allow unauthorized access.
- We validate our approach using real-device testing services and further compare it with the state-of-the-art method on physical devices, demonstrating its effectiveness and advantages.
- Applying our approach to large-scale custom ROMs, we uncover thousands of system properties and settings that store non-resettable identifiers, many of which lack proper protection. We also investigate the causes and characteristics of these vulnerable implementations. Our findings have been reported to the vendors and confirmed.

The source code of our tool is publicly available [24].

II. BACKGROUND

A. Non-Resettable Device Identifiers

In Android, non-resettable device identifiers are unique identifiers that cannot be altered or reset by the user, providing a consistent means of identifying a device. In our study, we consider the seven most commonly used non-resettable device identifiers, which are officially documented and frequently referenced in numerous researches [3,10,15,25,26].

- *International Mobile Equipment Identity (IMEI):* A fixed-length decimal digits assigned to mobile devices to identify them on a cellular network.
- *Mobile Equipment Identifier (MEID):* similar to the IMEI but used in CDMA phones.
- *International Mobile Subscriber Identity (IMSI):* a unique identifier assigned to the user of a cellular network, stored on the SIM card.
- *Integrated Circuit Card Identifier (ICCID):* a unique identifier for a SIM card itself.
- *Device serial number:* a unique identifier assigned by the device manufacturer.
- *WiFi MAC address:* a unique identifier for the network interface card of a device.

[1]A system ROM is a firmware image that contains the operating system and associated software for Android devices.

TABLE I: SELinux Key Concepts Explanation.

SELinux Key Concepts	Format	Example
Property Context	property_name user:role:type:sensitivity[:categories]	**(a)** xxx.xxx.xxx.imei1 u:object_r:system_id_prop:s0
Policy Rule	allow source_type target_type:class permissions	**(b)** allow system_app system_id_prop (file (read getattr map open))
		(c) allow radio system_id_prop (property_service (set))
	allow type_attribute_set type_attribute_set:class permissions	**(d)** allow appdomain extended_core_property_type (file (read getattr map open)))
Type Attribute Set	typeattributeset attribute_name [(and type1 ...)] [(not type1 ...)]	**(e)** typeattributeset extended_core_property_type (system_id_prop ...)
Keyword expandtypeattribute	(expandtypeattribute attribute_name <true, false>)	**(f)** (expandtypeattribute extended_core_property_type true)

- *Bluetooth MAC address:* a unique identifier for the Bluetooth module of a device.

The broad scope and stable nature of these identifiers are essential for user tracking, but they also pose significant privacy risks. Thus, they are no longer permitted for use by third-party apps after Android 10.

B. System Properties, Settings, and Access Control

System properties in Android are key/value pairs used to manage system configuration [27]. These properties are managed by the property service, with lightweight access via system APIs or commands and direct updates through the property service. **System settings** are key/value pairs that store device preferences and are accessible system-wide via system APIs. They are managed by the `SettingsProvider` and organized into *System*, *Secure*, and *Global* categories, each with distinct scopes and permission requirements.

System properties and settings are accessible to all entities within the system by default. However, since they often store sensitive information like identifiers, access control is necessary for certain properties and settings. Security-Enhanced Linux (SELinux) [28] enforces mandatory access control (MAC) [29] over system properties following a default-deny principle. Key SELinux concepts are illustrated in Table I through representative examples. Specifically, SELinux adopts a type-based policy model, where each system object and process is assigned a type attribute defining its security domain. For system properties, property contexts associate specific types with individual properties, as shown in Table I example (a). Access decisions are governed by policy rules that define which source types (e.g., system apps or services) may perform specific actions (e.g., read or write) on target types (examples (b)-(d)). SELinux also allows grouping multiple types under a shared attribute using type attribute sets (example (e)). The `expandtypeattribute` keyword determines whether rules for an attribute apply to all its member types (example (f)). These policy rules are written in Type Enforcement (TE) files during development and compiled into binary policy files. Since Android 8.0, SELinux policies have been expressed in the Common Intermediate Language (CIL) format to support better modularity and maintainability.

Android also enforces access control policies over system settings. Write access requires permissions such as `WRITE_SETTINGS` and `WRITE_SECURE_SETTINGS`. For read access, prior to Android 12, all system settings were readable by third-party apps unless explicitly marked with "@SystemApi," which restricts access to system apps. Starting with Android 12, AOSP introduced the "@Readable" annotation [30] to provide finer-grained control: settings without this annotation are no longer accessible to non-system apps. However, settings introduced by privileged apps but not defined in the `Settings` class lack proper access control, both before and after Android 12.

III. IDRadar

To thoroughly understand the covert access channels to non-resettable device identifiers in large-scale custom Android systems, we design IDRadar, a static analysis-based framework that identifies sensitive and vulnerable system properties and settings exposing such identifiers. Figure 1 illustrates the overall architecture of the framework. IDRadar operates in three main phases: (1) Static Analysis: locates all usage of system properties and settings in custom ROMs through control- and data-flow analysis (§III-A); (2) Filtering and Confirmation: applies heuristic rules to identify candidate properties and settings that are likely to store sensitive identifiers, followed by manual verification (§III-B); (3) Vulnerability Detection: analyzes access control policies to find cases where third-party apps can access these identifiers without proper permissions (§III-C). The reliance on static analysis, along with a largely automated and scalable design, makes IDRadar well-suited for large-scale analysis across diverse custom Android ROMs. We next elaborate on each phase in detail.

A. Static Analysis

To discover custom system properties and settings, a straightforward idea is to identify all locations where they are defined. In practice, however, this is challenging because such definitions may appear across multiple layers of the Android system (e.g., Linux kernel, system libraries, frameworks, and pre-installed apps) and lack a consistent format. Therefore, we adopt an alternative approach: we focus on the usage of system properties and settings and analyze their content using contextual information. This approach is based on the assumption that custom properties and settings are typically introduced to support specific customization functionalities. In addition, systems are unlikely to define such properties or settings without actually using them. Hence, analyzing their usage provides an effective way to discover custom system properties and settings. We acknowledge that this assumption is inherently difficult to validate because it is challenging to obtain the ground truth of system properties and settings that actually exist on a device, a point we further discuss in §V-B.

To implement this approach, our analysis begins by summarizing the access methods for properties and settings, then locates their usage in the codebase, and finally examines their names and surrounding code context for further evaluation.

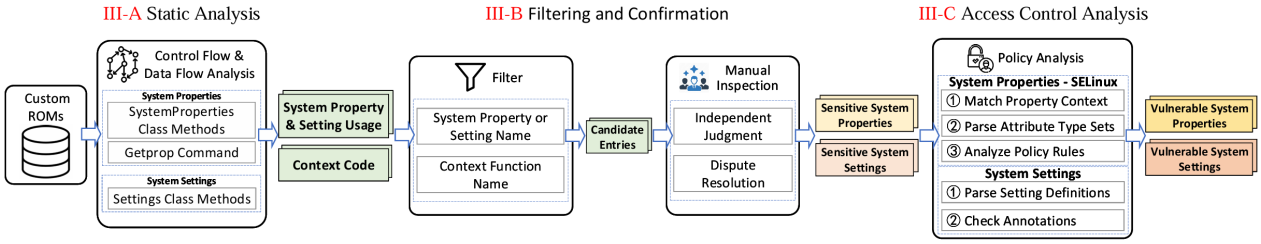

Fig. 1: The Overview of IDRADAR.

1) Methods of accessing system properties/settings: Although finding usage is a relatively feasible method, it is not trivial, especially when it comes to locating the usage of system properties. Unlike system settings, which can be accessed through well-documented methods, system properties are often accessed via "non-SDK methods[2]" or system commands. Therefore, accessing system properties is more variable. To comprehensively identify the usage of system properties, we first conduct a pilot study to collect the various property access methods. In this study, we automatically extract DEX files from APKs and JARs in the custom ROMs and apply the `strings` command [31] to retrieve all embedded strings. From these strings, we select candidates with recognizable property prefixes such as "ro.", "persist.", and "vendor." [15]. For each candidate, we manually reverse-engineer the corresponding code using a decompiler to verify whether it represents an actual property access and further summarize the observed access patterns. In total, this pilot study analyzes 210 samples, enabling us to systematically understand how system properties are accessed in custom systems.

Based on the pilot study, we summarize the methods for accessing system properties and settings. For brevity, we organize them into patterns (a)–(f), as illustrated in Figure 2. In addition, the distribution of each access pattern is reported in parentheses within the sub-caption of Figure 2, including both frequency counts and proportions.

Overall, system properties are accessed mainly through two approaches: non-SDK methods in the `android.os.SystemProperties` class (patterns (a)–(d), accounting for 94.4% of all observed accesses) and the `getprop` system command (pattern (e), 5.6%). In customized systems, only a subset of system components (e.g., system services) can directly invoke methods of the `SystemProperties` class (pattern (a), 62.6%[3]). Other components (e.g., pre-installed apps) instead rely on Java reflection, which typically follows a three-step process (pattern (b), 29.2%): (1) dynamically loading the `SystemProperties` class with `Class.forName` (Statement ❷); (2) obtaining the `get` method using `Class.getMethod` (Statement ❸); and (3)

[2]Non-SDK methods encompass those not included in the public SDK, often belonging to internal APIs or used only in system-level apps only, leading to variable access behaviors.

[3]Although only a small number of system components can apply this pattern, these components exhibit significantly more property accesses than others, making this pattern appear most frequently in our study.

```
String propertyValue = SystemProperties.get(propertyName);                    ❶
```
(a) Access System Properties via Direct Method Invocation (14,128 - 62.6%)

```
Class<?> systemPropertiesClass = Class.forName("android.os.SystemProperties");   ❷
                                                                  ①
Method getMethod = systemPropertiesClass.getMethod("get", String.class);      ❸
                                                  ②
String propertyValue = (String) getMethod.invoke(null, propertyName);         ❹
```
(b) Access System Properties via Normal Reflection (6,580 - 29.2%)

```
private static Method cachedMethod = Class.forName("android.os.SystemProperties")
                                                  .getMethod("get", String.class); ③
                                                             ④
String propertyValue = (String) cachedMethod.invoke(null, propertyName);      ❺
```
(c) Reflection with Cached Method Field (345 - 1.5%)

```
private static Method helper() {
    return Class.forName("android.os.SystemProperties").getMethod("get", String.class);
}                                                  ⑤                  ⑥

String propertyValue = (String) helper().invoke(null, propertyName);          ❻
```
(d) Reflection with Helper Method (235 - 1.0%)

```
Process process = Runtime.getRuntime().exec("getprop " + propertyName);       ❼
BufferedReader reader = new BufferedReader(new                    ⑦
    InputStreamReader(process.getInputStream()));
String propertyValue = reader.readLine();
reader.close();
```
(e) Access System Properties via System Command (1,274 - 5.6%)

```
String settingValue = Settings.Global.getString(getContentResolver(), settingName);
                    = Settings.System.getString(getContentResolver(), settingName); ❽
                    = Settings.Secure.getString(getContentResolver(), settingName);
```
(f) Access System Settings via SDK Methods

Fig. 2: Access Patterns of System Properties and Settings.

invoking this method with the target property name via `Method.invoke` (Statement ❹). The invocation returns a string representing the property value.

While the straightforward three-step reflection pattern, in which all operations appear explicitly in a single method, is common, our pilot study also revealed two variations in reflection usage: pattern (c), reflection with cached method field (1.5%), where the first two reflection steps are executed during static initialization and the resulting `Method` object is stored in a static field for later use; and pattern (d), reflection with helper method (1.0%), where one or two steps are encapsulated in helper methods that return the `Class` or `Method` object.

In contrast to the diverse access patterns for system properties, accessing system settings is more uniform, because Android provides SDK methods for this purpose. Such accesses primarily occur through methods in the `android.provider.Settings` class, as illustrated in pattern (f).

2) Identifying system properties/settings: After clarifying the access methods of system properties and settings, our next goal is to locate their usage in the codebase, along with the

contextual code that can reveal their content. To this end, we perform static analysis with Soot [32] on all APK and JAR files, decompiling them into Jimple, the intermediate representation used by Soot. We then analyze each method to identify accesses to properties and settings, focusing on method invocations and field references.

For method invocations, we check whether the invoked methods are related to accessing system properties or settings. These include calls to `SystemProperties` (Statement ❶ in Figure 2), `Settings` (Statement ❽), reflection methods such as `Class.forName` (Statements ❷–❹), and command execution methods like `Runtime.exec` (Statement ❼). In addition, we examine whether a method's return type is `Class` or `Method` (e.g., the `helper` method in Statement ❻), corresponding to pattern (d). For field references, we check whether the field type is `Method` (e.g., the `cachedMethod` field in Statement ❺), corresponding to pattern (c).

Reflection and system command execution can serve many purposes beyond accessing system properties. Therefore, we further analyze whether the invoked reflection methods or executed commands are related to property access. To this end, we consider three cases: (1) For normal reflections and command executions, we check whether the invoked methods (e.g., Parameters ① and ② in Figure 2) belong to `SystemProperties` or whether the executed commands are `getprop` (e.g., Parameter ⑦). (2) For patterns (c) and (d), we analyze the field definition process in class initializers (Parameters ③ and ④) and the return-value definition process in helper methods (Parameters ⑤ and ⑥) to determine whether they reference methods in `SystemProperties`. (3) If the parameters of reflections or executed commands (Parameters ①–⑦) are not local variables but are instead passed as method parameters, we construct the interprocedural control-flow graph (ICFG) to trace the call sites of the method and retrieve the corresponding parameter values. This enables us to determine whether they are related to property access.

3) Analyzing system properties/settings: Next, for confirmed system property and setting accesses, we analyze the property and setting names used as parameters in key methods (e.g., "propertyName" and "settingName" in Figure 2). We handle five cases during this process: (1) If the name is a string constant, we directly extract its value. (2) If it is a method parameter, we use the ICFG to trace the call sites and recover the parameter values. (3) If it is built through string concatenation, we collect the involved substrings and reconstruct the complete name. (4) If it is a field, we examine the class initializer to retrieve its value. (5) If it is stored in an array, we extract all array elements as potential names.

After identifying the names of accessed properties and settings, we extract their contextual code for further analysis. Using the ICFG, we backtrack from the call sites to construct the full method call chain and the related source code.

B. Filtering and Confirmation

Through static analysis, we extracted numerous system properties and settings from the custom ROM, along with their contextual code. We then aim to identify those containing the seven types of non-resettable device identifiers defined in §II-A, which we refer to as **sensitive system properties and settings** throughout this paper for clarity. To this end, we apply two simple yet effective heuristics: (1) selecting properties and settings whose names contain identifier-related keywords (e.g., IMEI, ICCID, DeviceID), as these names in systems are generally descriptive and unobfuscated; (2) identifying context methods with identifier-related names (e.g., "getIMEI", "getWifiMac") and inferring that the accessed properties/settings correspond to those identifiers.

```
public String getIMEI(Context context) {
    String imei = SystemProperties.get( "xxx.xxx.xxx.imei" );
    Log.d( "Get IMEI from property xxx.xxx.xxx.imei" );                    ❶
    if (!isValid(imei)) {
        TelephonyManager systemService = context.getSystemService( "phone" );
        imei = systemService.getDeviceId();                                ❷
        Log.d( "Get IMEI from system service" );
    }
    return imei.
}
```

Fig. 3: Key Information for Identifying Property Content.

This process narrows down the system properties and settings likely to contain sensitive identifiers. We then manually inspect the contextual code to confirm which entries store non-resettable identifiers. Such code often provides useful clues that help determine the content of properties and settings, specifically: (1) Method names, which can reveal the intended functionality and thus imply the type of identifier being accessed. For example, in Figure 3, the method `getIMEI` returns a system property value, suggesting that this property stores the IMEI. (2) Log messages, which often expose retrieval actions and associated identifiers directly. In Figure 3 ❶, the log message "reading IMEI from property" following a property access provides explicit evidence of its content. (3) Functional behavior, which reflects code logic that helps reveal the content of properties and settings. In Figure 3 ❷, the code first reads a system property and, if the result is invalid, falls back to retrieving the IMEI from a system service. This behavior indicates that the property stores the device's IMEI, consistent with the system service. In practice, the presence of any such clues can help us determine the content of the entries and verify whether they store non-resettable identifiers.

While performing this manual analysis, we found that some properties and settings were accessed only by third-party pre-installed apps, which attempted to retrieve identifiers from multiple sources. This behavior suggests that these apps are uncertain whether such entries actually exist in the current system and therefore attempt all possible options. Such speculative access patterns could lead to false positives. To mitigate this issue, we use the `strings` command to check whether the identified properties or settings also appear in system components, including non-JAR/APK components outside the scope of our static analysis. If a property or setting name is found in any system component, we treat it as valid evidence of its existence, thereby reducing potential false positives.

C. Access Control Analysis

After identifying sensitive system properties and settings that contain non-resettable device identifiers, we analyze their access control policies determine which lack effective protection. We refer to these unprotected entries as **vulnerable system properties and settings** throughout this paper. As noted in §II-B, access control for system properties is enforced by SELinux. We begin by locating the relevant `property_contexts` files in custom ROMs, which define the property context for each property. We then examine these files to determine the contexts assigned to properties storing sensitive identifiers, focusing on the "type" field (e.g., system_id_prop" in Table I). Property names in these files may include wildcards (e.g., "ro.*"), which can lead to multiple matching entries. In such cases, SELinux selects the most specific match, typically the one with the longest prefix that matches the property name.

After identifying a property's context, we analyze its associated type attribute sets and policy rules. We collect all relevant policy and binary/CIL files from the ROMs and conduct a two-pass analysis of SELinux rules. In the first pass, we extract type attribute set definitions, including uses of the `expand typeattribute` keyword, and resolve each set. Concrete types and non-expandable sets are directly identified, while expandable sets are recursively resolved. In the second pass, we examine all policy rules, identifying those whose target types match a sensitive property either directly or through an attribute set. We then analyze its source types to determine which entities, such as "system_app" in Table I, are allowed to access them. Since third-party apps are typically labeled as "untrusted_app", any rule granting access to this type indicates that the property is accessible to third-party apps.

To analyze access control policies for system settings, we examine the `android.provider.Settings` class. Using Soot, we analyze all fields from the `Settings.Global`, `Settings.System`, and `Settings.Secure` inner classes, extracting their definations and annotations such as "@Readable" and "@SystemApi". Each field corresponds to a system setting key. By analyzing these definitions and annotations, we determine which settings are accessible to third-party apps under different Android versions.

IV. Results

In this section, we present large-scale measurements of real-world custom Android systems. We analyzed the prevalence of covert channels used to access non-resettable identifiers in these systems (RQ1), and assessed how well such access is protected by permission mechanisms (RQ2). We also evaluated the effectiveness of IDRadar in detecting vulnerable system properties and settings (RQ3), and examined the vulnerabilities in vendor-specific customization practices (RQ4). The specific research questions are outlined as follows:

- RQ1 How prevalent is the introduction of custom system properties and settings for accessing non-resettable identifiers in real-world custom systems?
- RQ2 To what extent do custom Android systems lack proper access control over system properties and settings, and what causes these protection gaps?
- RQ3 How effective is IDRadar in identifying vulnerable system properties and settings, and how well does it perform compared with state-of-the-art approaches?
- RQ4 Are vulnerabilities repeated in vendor-specific customizations across different devices?

A. Evaluation Setup

1) Customized System Dataset: To support our large-scale investigation, we collected custom Android ROMs from the public project Android Dumps [33], which hosts a wide range of Android stock ROMs. In October 2023, we collected all available ROMs from this source, resulting in a dataset of 1,814 ROMs across 250 vendors. Detailed information on vendors, ROM counts, and Android API versions is summarized in Table II. Our dataset consists mainly of ROMs from major vendors such as Samsung, Xiaomi, Huawei, and Lenovo, which comprise the majority, along with smaller vendors such as Gionee and Blackview, typically represented by only 2–3 ROMs each. It also includes Google's ROMs for Pixel devices, which contain Google's own system customizations. The ROMs in the dataset cover Android versions from 4.0.3 to 14, with build dates ranging from 2013 to 2023. It is worth noting that although systems prior to Android 10 permitted third-party access to non-resettable identifiers, such access still required user-granted permissions. Covert channels, however, enable unauthorized access to identifiers, posing a greater privacy risk. Therefore, analyzing these channels remains meaningful even for pre-Android 10 systems. This diverse dataset enables a comprehensive exploration of covert channels used to access non-resettable device identifiers in custom Android systems. Notably, the ROMs from Android Dumps are already parsed into file collections rather than packaged as images, allowing direct analysis without additional unpacking.

2) Experimental Environment and Performance: Our experiments were conducted on a server equipped with an AMD EPYC 7713 CPU and 500 GB of RAM. We evaluated the efficiency of IDRadar in performing static analysis on custom ROMs. On average, this phase took approximately 110 minutes per ROM, with each parallel task consuming 20 GB of memory under 8-way parallelism.

B. RQ1: Sensitive Custom System Properties/Settings

1) Identification of Sensitive Properties and Settings: Applying IDRadar to all collected custom ROMs, we successfully analyzed over 600,000 APK and JAR files within these ROMs. Our analysis initially produced over 10 million usage cases of properties and settings, which were reduced to approximately 30,000 after the filtering process described in §III-B. The following manual analysis was conducted independently by two authors with relevant expertise, with a third resolving any disagreements. Consensus between the two was accepted; otherwise, the third author mediated the decision. This verification process took three days. Regarding

TABLE II: The Distribution of Our Custom System Dataset and Summary of Our Results.

Brand	# ROMs	Android Version									# Sensitive Properties	# (%) Vulnerable Properties	# Sensitive Settings	# (%) Vulnerable Settings	# (%) Sensitive Devices	# (%) Vulnerable Devices
		Pre-v6	v7	v8	v9	v10	v11	v12	v13	v14						
Alps	26	3	3	7	4	5	0	3	1	0	94	40 (43%)	24	5 (21%)	10 (38%)	7 (27%)
Asus	55	1	6	10	11	11	3	3	10	0	36	22 (61%)	35	3 (9%)	12 (22%)	6 (11%)
Digma	25	0	1	20	4	0	0	0	0	0	20	3 (15%)	35	0 (0%)	25 (100%)	1 (4%)
Huawei	29	2	1	0	5	7	5	9	0	0	6	1 (17%)	5	1 (20%)	5 (17%)	1 (3%)
Lenovo	99	21	9	8	26	12	13	6	4	0	529	337 (64%)	182	51 (28%)	85 (86%)	75 (76%)
Meizu	40	10	6	8	6	2	4	0	3	1	218	125 (56%)	142	44 (31%)	39 (97%)	37 (93%)
Motorola	144	9	11	17	24	21	13	33	16	0	416	162 (39%)	146	55 (38%)	118 (82%)	113 (78%)
Nokia	88	4	1	5	17	30	10	9	12	0	363	266 (73%)	100	36 (36%)	49 (56%)	40 (45%)
Nubia	29	0	3	3	8	3	5	4	3	0	78	46 (59%)	87	25 (28%)	29 (100%)	21 (72%)
Oneplus	45	2	0	0	3	8	17	8	7	0	195	106 (54%)	78	47 (60%)	38 (84%)	36 (80%)
Oppo	64	6	2	8	9	10	5	6	18	0	384	229 (60%)	145	25 (17%)	61 (95%)	59 (92%)
Poco	22	0	0	0	0	3	8	6	5	0	363	69 (19%)	120	69 (56%)	22 (100%)	22 (100%)
Qti	42	0	0	0	2	2	6	28	4	0	326	132 (40%)	173	93 (54%)	41 (98%)	40 (95%)
Realme	65	0	0	1	8	19	11	10	15	1	396	192 (48%)	137	37 (27%)	65 (100%)	65 (100%)
Redmi	59	0	0	0	0	7	10	11	26	5	673	124 (18%)	208	100 (48%)	59 (100%)	54 (92%)
Samsung	158	20	11	13	28	27	25	15	19	0	534	226 (42%)	524	212 (40%)	154 (97%)	134 (85%)
Xiaomi	146	9	12	24	11	25	26	4	29	6	1598	441 (28%)	647	281 (43%)	137 (94%)	116 (79%)
Zte	28	2	2	3	11	5	3	2	0	0	153	116 (76%)	188	57 (30%)	23 (82%)	15 (54%)
Other (232)	650	71	42	198	148	80	65	30	14	2	1810	840 (46%)	644	195 (30%)	434 (67%)	270 (42%)
Total (250)	1814	160	110	325	325	277	229	187	186	15	8192	3477 (42%)	3620	1336 (37%)	1406 (78%)	1112 (61%)

Note: We list separately the 18 brands with more than 20 ROMs; the remaining 232 are grouped together. ROMs from Android 6 and earlier are combined due to limited samples. Percentages in the Vulnerable Properties and Vulnerable Settings columns are based on the number of vulnerable vs. sensitive entries, while those in Sensitive Devices and Vulnerable Devices are based on total device count.

agreement, contextual clues allowed clear determinations in most cases, and the two authors disagreed in only 3% of the cases. The inter-rater reliability, measured by Cohen's kappa, was 0.966. During this manual verification process, we noticed that ROMs from the same brand often contained repeated cases, including identical system property or setting names and contextual code. This suggested that these properties and settings were reused across systems, allowing us to skip redundant reviews and significantly reduce the verification workload. As a result, we confirmed 8,192 system property cases and 3,620 system setting cases containing non-resettable identifiers, corresponding to 272 unique system properties and 104 unique settings (noting that the same property or setting may appear across multiple ROMs). In our dataset, 1,406 (77.5%) custom ROMs have at least one such sensitive system property or setting, indicating the widespread presence of covert identifier exposure in custom Android systems. Detailed results are presented in Table II (columns marked *Sensitive*).

2) Evolution Across Android Versions: We also examined how sensitive system properties and settings have evolved across different Android versions. As shown in Figure 4, their average number of sensitive entries in custom ROMs gradually increased from earlier versions, with a notable spike in Android 10-11. This surge likely corresponds to the introduction of stricter platform restrictions, particularly the complete prohibition of third-party apps from accessing non-resettable identifiers in Android 10. A slight decline is observed in later versions, possibly reflecting growing regulatory pressure from both the public and academic communities, as well as improved awareness among OEMs. This trend supports our hypothesis that system changes undermine stable tracking via official APIs, prompting supply chain actors to adopt covert channels instead. Note that our dataset contains only 15 custom ROMs for Android 14, which may have skewed the results for that version, potentially making it an outlier.

3) Supply Chain Actors Using Covert Channels: We further examine which supply chain actors leverage covert channels to access device identifiers and for what purposes. These

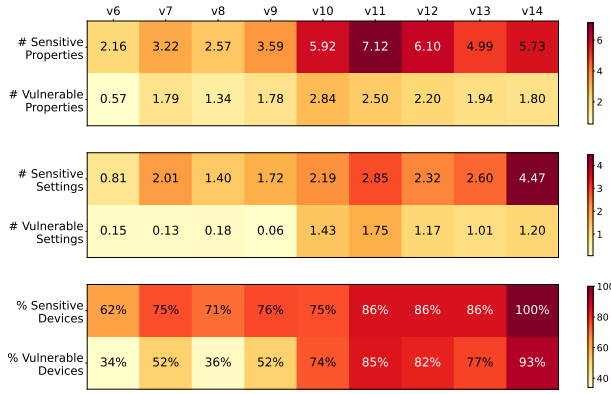

Fig. 4: The Average of System Properties/Settings and Percentage of Devices across Different System Versions.

actors can be identified by analyzing the package names of the code that accesses sensitive system properties and settings. We focus on packages that occur frequently for a detailed case study. Our analysis reveals that both custom system developers (e.g., Oppo [34]) and hardware providers (e.g., MediaTek [35]) access these properties within system components to monitor device status. For example, MediaTek's telephony module reads the IMSI or ICCID from system properties to determine whether a SIM card is present. In addition, various SDKs embedded in system apps also utilize these properties and settings. These include vendor-provided SDKs such as Xiaomi's [36] analytics and crash reporting tools, Lenovo's [37] push service, and third-party SDKs like Baidu's [38] map SDK and Alibaba's [39] security SDK. These identifiers are used for device recognition, usage tracking, crash reporting, and transmitting related data to remote endpoints.

Answer to RQ1: *Sensitive system properties and settings that store non-resettable identifiers are widely present*

in custom systems across various brands and Android versions, with a total of 11,812 cases identified in 1,814 custom ROMs. Their introduction is likely driven by the need for convenient access to identifiers in response to platform-level restrictions.

C. RQ2: Vulnerable Implementations

1) Vulnerability Distribution: We then analyzed the access control policies of these sensitive system properties and settings (as described in §III-C) to determine whether they are properly protected. Among the 8,192 system properties and 3,620 system settings, we identified 3,477 (42.4%) properties and 1,336 (36.9%) settings lacking proper access control, allowing unrestricted access by third-party apps. At least one such vulnerable case was found in 1,112 (61.3%) custom ROMs. Detailed results are presented in Table II (columns marked *Vulnerable*). Notably, we found no such issues in Google's ROMs, reflecting Google's strong code security practices, a finding consistent with prior work [18,19]. The version-wise trend of vulnerable entries closely mirrors that of sensitive ones, as shown in Figure 4. Before Android 10, the smaller number of sensitive properties and settings likely made it easier to implement correct access control. After Android 10, as their number grew, enforcing access control became more complex and error-prone.

2) Root Causes of Vulnerabilties: We further conducted a manual examination of these vulnerable cases and summarized several common causes underlying the insecure practices.

(*i*) Overlooked System Properties and Settings. We believe the most direct cause of vulnerable access control is developer oversight, where appropriate policies were not applied to sensitive system properties and settings. For instance, in BRAND-A's custom ROM (Android 14), we reported four vulnerable system properties "xx.xx.xx.imei1", "xx.xx.xx.imei2", "xx.xx.xx.meid", and "xx.xx.xx.sn", which were later confirmed by the vendor. Our analysis of all property contexts in this ROM showed that the most matching context for these properties is "*", the default property type. This default context matches any system property and grants access to third-party apps. In this case, it appears the developer forgot the required context definition, resulting in a vulnerability. In total, we found 844 similar cases where sensitive properties were matched only by the default context. Likewise, 95% of the vulnerable system settings we identified were not defined in the `Settings` class, allowing unrestricted access by third-party apps due to missing access control annotations.

(*ii*) Overly Complex Access Control Rules. Extensive customizations can lead to a large number of SELinux rules in custom systems. For example, in the custom ROM of BRAND-B (Android 14), there are over 2,500 property contexts and more than 50,000 lines of specific policy rules. We have reported two vulnerable system properties "xxx.xxx.xxx.btmac" and "xxx.xxx.xxx.wifimac" identified in this ROM, which was also confirmed by device vendors. These properties belong to "radio_prop" type, which is included in six type attribute sets and linked to dozens of policy rules. One of these rules allows "untrusted_app" to access it, resulting in the vulnerability. Therefore, we believe that overly complex access control rules contribute to such issues. On the one hand, as in the case of the above example, involving too many type attribute sets can increase the likelihood of misconfiguration. This is because a mistake in any sets can lead to access control errors for the system property. On the other hand, an excessive number of rules degrade the auditability of access control policy, making it more difficult to identify existing access control flaws.

(*iii*) Covert Supply Chain Actors. A wide range of supply chain actors contribute to system customization, and some covertly introduce sensitive system properties or settings during this process. Access control policies are typically designed around custom properties and settings added by major actors, such as device vendors or hardware providers, while those introduced by covert actors may be overlooked, leading to potential vulnerabilities. For example, Baidu [38] introduces two sensitive system settings "xxx_xxx_i", which stores the device's plaintext IMEI, and "xxx.xxx.deviceid", which holds an encrypted key-value pair containing the IMEI and an identifier derived from the `android_id` [40]. The encryption uses AES in CBC mode, but with a key hard-coded. These two settings are widely embedded in custom systems. Our analysis shows that access to these settings primarily occurs within Baidu SDKs (e.g., package name "com.baidu.lbsapi"), commonly integrated into system apps such as app stores, maps, browsers, and weather apps. We identified 3,120 apps accessing these settings. Their widespread use, coupled with the absence of access control in all systems we analyzed, significantly impacts our results (Table II and Figure 4).

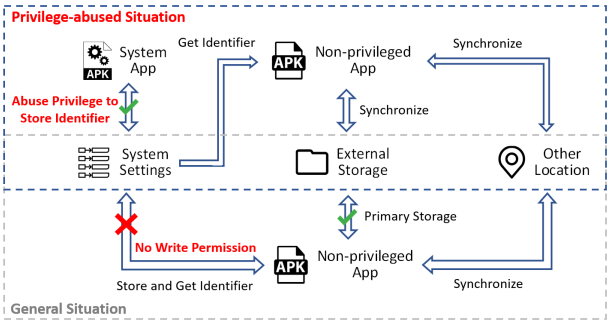

Fig. 5: Identifier Handling Process of the Baidu SDK.

More critically, we found that the same Baidu SDKs (with identical package names) had been previously reported in related work [41], which showed the SDKs abused Android's external storage to share device identifiers across apps, enabling unauthorized user tracking. The authors assumed that non-privileged apps lacked permission to write to system settings, and thus concluded that external storage was the primary location for storing identifiers. Our findings extend this conclusion by showing that these SDKs are also embedded in system apps on customized systems. In this context, the

SDKs inherit privileged permissions and misuse them to write identifiers directly into system settings (see Figure 5). Since reading system settings requires no permission, even the SDKs embedded in non-privileged apps can retrieve these identifiers without restriction. Compared to external storage, system settings provide a more stable and stealthy storage location. Moreover, as system settings are intended for internal configuration, Android offers no user-facing interface to view or manage them, making it extremely difficult for users to detect or remove stored identifiers. This tracking mechanism, which relies on privileged system components to enable unauthorized access by non-privileged apps, poses a significant threat to user privacy and undermines transparency.

Answer to RQ2: *Vulnerable system properties and settings are widespread in custom Android systems, with 4,813 cases identified, accounting for 41.4% of all sensitive entries. These vulnerabilities are primarily caused by developer oversight, extensive system customization, and the involvement of covert supply chain actors.*

D. RQ3: The Effectiveness of IDRadar

1) Validation on Real Devices: We next verify the precision of the vulnerable system properties and settings we identified by generating Proof of Concepts (PoCs) on real devices. Due to limited device availability across brands and models, we leveraged remote real-device testing services offered by vendors and commercial platforms (e.g., Alibaba Cloud [42]). These services provide remote desktop access to a wide range of devices, allowing users to upload and run test apps. However, in our dataset, there is only one ROM for each device, corresponding to a specific system version. The remote testing services may not provide the exact device we need, and even if they do, the system version might differ from our dataset. As a result, we could not generate PoCs for all vulnerabilities. Alternatively, we were able to find devices in the remote testing service whose models and Android versions matched those in our dataset, though their build dates may have differed. We acknowledge that such build date differences may affect the presence or accessibility of properties and settings (e.g., due to changes in applied access control policies). To minimize this impact, we selected 32 devices with build dates as close as possible to those in our dataset as the test devices, thereby reducing potential discrepancies and their influence on the validation results. We then used these devices to evaluate our results. Specifically, we developed a test app without any permission requests that simulates a third-party app and attempts to retrieve all system properties and settings. We then manually compared the retrieved values with device identifiers shown in the settings app, confirming they contained non-resettable identifiers.

In summary, our approach identified 130 vulnerable properties and settings across 32 devices, and we successfully generated PoCs for 102 of them. All 102 were verified to store non-resettable device identifiers. The remaining 28 cases primarily correspond to a single system setting, "xxx_device_mac", which is accessed by MediaTek's [43] telephony component `MtkTeleService`. Our static analysis indicates that `MtkTeleService` attempts to read the vulnerable setting, but the setting was not present at runtime during testing. Manual inspection further revealed that the code is intended to retrieve the MAC address on TV devices, as suggested by a log message reading "get TV mac exception". It is therefore likely that this setting is only triggered on certain TVs, whereas our test devices were smartphones. This case demonstrates a limitation of our static analysis: it may flag entries that are activated only under specific runtime conditions. Overall, based on the dynamic evaluation, we estimate the precision of our approach to be 78.5% (102/130).

2) Comparison with State-of-the-art: We also compare with U2-I2 [15], the most closely related work. First, in terms of research scale, U2-I2 covered only 13 devices and identified 30 vulnerable system properties and settings. In contrast, our dataset and findings are two orders of magnitude larger, offering broader coverage across time and device diversity, and enabling a more comprehensive analysis of non-resettable identifiers. Second, we tried to compare the capability of these two approaches for analyzing the same system. However, U2-I2's open-source artifact [44] anonymized the test devices, making direct comparison with their results infeasible. Additionally, U2-I2 relies on factory resets, which cannot be performed on remote testing platforms. Ultimately, we gathered four physical devices whose ROMs are included in our dataset and conducted dynamic testing using the U2-I2 tool. Following its guidelines, we used the "GETID" function to test for vulnerable properties and settings before and after a factory reset. We found the process to be labor-intensive. Since a factory reset was required, we first had to restore the rooted test devices to prevent them from becoming bricked. Moreover, several key steps, including the factory reset, re-enabling ADB debugging, identifying relevant properties and settings, and comparing results, could not be automated. Notably, many entries remained unchanged after the reset, requiring manual inspection of the settings app to confirm whether they corresponded to non-resettable identifiers.

As a result, U2-I2 identified three non-resettable identifiers in system properties and settings across the four devices. Our method detected all three and additionally uncovered two more identifiers in system settings, namely "xxx_sim_imsi" and "xxx_sim_iccid", which were present on two devices. These settings are associated with a mobile data feature and only appear after specific actions are performed on the device. Due to the lack of complete feature setup, U2-I2 failed to detect them. This highlights a key limitation of U2-I2 in identifying issues that depend on specific preconditions, whereas our static approach can uncover such cases. We did, however, encounter one potential false positive (the aforementioned "xxx_device_mac"), which was flagged by our method but not observed during dynamic testing. In summary, our method reproduces U2-I2's findings while additionally revealing vulnerabilities that U2-I2 misses.

Answer to RQ3: *Dynamic testing on real devices demonstrates the high precision of our approach. In addition, compared to existing work, it supports a wider range of devices and uncovers additional vulnerabilities on the same custom system that U2-I2 fails to detect.*

E. RQ4: Brand-Wide Recurring Vulnerabilities

As noted in IV-B, we observed that the same vulnerabilities frequently recur across multiple devices from the same brand. Motivated by this observation, we extended our testing to additional new devices–whose ROMs were not included in our dataset–from previously studied brands, using remote real-device testing services. Specifically, we attempted to determine whether the previously identified vulnerable properties and settings from the same brand persist on these new devices. This helps us assess the generalizability of vulnerabilities introduced through vendor-specific customization.

TABLE III: The Device Information of Extended Evaluation.

Brand	# Devices	Android Version										# VP	# VS
		v6	v7	v8	v9	v10	v11	v12	v13	v14	v15		
Huawei	11/29	0	0	3	3	20	0	3	0	0	0	0	22
Motorola	7/7	1	3	0	0	0	3	0	0	0	0	18	11
Oppo	31/53	0	0	0	2	5	10	6	17	13	0	4	56
Oneplus	7/11	0	0	0	0	0	0	0	0	9	2	0	15
Samsung	21/35	0	2	2	3	3	0	10	9	6	0	12	50
Vivo	97/110	0	0	4	2	7	24	18	28	26	1	36	183
Xiaomi	9/20	0	0	0	0	0	0	0	12	8	0	4	9
Redmi	13/23	0	0	0	0	0	1	2	17	3	0	10	12
Other (14)	20/26	1	6	3	4	4	4	3	0	0	1	76	42
Total (22)	216/314	2	11	12	14	39	42	42	83	65	4	160	400

Note: We list separately the 8 brands with more than 5 devices. The Devices column shows the total tested and vulnerable ones. VP and VS indicate the number of vulnerable properties and settings.

As summarized in Table III, our dynamic test covered 314 new devices across 22 brands. We identified a total of 160 vulnerable system properties and 400 vulnerable settings, which could be accessed by third-party apps without any permissions. For example, among the 35 Samsung devices we tested, 21 contained recurring vulnerabilities. These results confirm our earlier observation that vulnerabilities tend to recur across devices of the same brand, likely due to code reuse within OEMs or upstream suppliers.

We promptly reported the confirmed issues to the respective vendors. Since vendors typically require corresponding proof (e.g., PoCs), we only reported vulnerabilities with tested PoCs on real devices running Android 12 or higher. We excluded older versions, as vendors usually prioritize security in recent products and older issues may already be patched. So far, two vendors have confirmed our findings, acknowledging 82 vulnerable system properties and two vulnerable settings, and have issued corresponding fixes in system updates.

Answer to RQ4: *Vulnerabilities frequently recur across different devices from the same brand, and additional testing without ROM access revealed many such repeated cases. We reported our findings to the respective vendors and received acknowledgment.*

V. Discussion

A. Mitigation

Clearer Access Control for System Settings. While Android provides relatively clear and comprehensive access control for system properties through SELinux, the protection mechanisms for system settings are far less well-defined. Official documentation offers limited information on system settings, and we found no clear explanation of their access control, with the only related details appearing in AOSP source code comments. We therefore recommend that Google strengthen access control for system settings and enforce stricter default policies for newly added settings to prevent third-party access.
Transparency in Covert System Properties and Settings. Although Google provides access control for major supply chain actors (e.g., vendors and ODMs) during SELinux rule configuration [45], many more actors participate in system customization. As a result, access control enforcers may be unaware of the introduction of certain properties or settings, such as the two sensitive settings added by the Baidu SDK. We recommend that all customization actors report newly added sensitive properties or settings to enforcers, enabling more comprehensive protection of non-resettable device identifiers.
Conducting Thorough Security Testing before Release. Verification shows that the ROMs of 1,516 devices in our dataset are supported by Google Play [46], indicating they passed Google certification and independent vendor testing. However, our findings reveal insufficient testing of sensitive system properties and settings. To address this, we propose a simple yet effective method, inspired by U2-I2 [15], based on the persistence of non-resettable identifiers. The process has two steps: (1) after system initialization, extensively use the device to trigger properties and settings and record their values; (2) perform a factory reset and repeat the process to identify unchanged entries. Since non-resettable identifiers persist across resets, this method can detect sensitive properties and settings, including those stored in encrypted form.

B. Limitations and Future Work

Our work has several limitations. First, we identified system properties and settings in custom systems by locating their usage in the codebase. While this approach is practical and scalable, it introduces certain constraints. Specifically, it cannot accurately determine whether a property or setting exists on a device, nor capture the specific runtime conditions under which entries are activated. Despite applying multiple methods to reduce false positives, some may still remain. In future work, we plan to explore multi-level system analysis (e.g., kernel, framework, apps) to directly identify property and setting definitions, thereby enabling more accurate analysis. Second, our reliance on static analysis and the large codebase of custom systems prevents us from obtaining the ground truth for all sensitive properties and settings. This limits our ability to verify whether all access behaviors were captured or whether any sensitive entries were missed. In evaluation, due to limited device availability and financial constraints, we

relied on remote real-device testing services to validate our findings. However, these services rarely provide devices that exactly match our dataset, which prevents us from generating PoCs for all vulnerabilities. In addition, the device states in such services are uncertain, making it difficult to ensure that they are fully initialized and that all sensitive properties and settings are exposed. To improve validation coverage, we plan to leverage crowdsourcing to access a broader range of devices. Third, our static analysis can be affected by obfuscation (e.g., when property or setting names are obfuscated). Although such cases were rare in our prior sampling (20 out of 30,000), they may still hinder accurate identification. We plan to explore symbolic execution to address obfuscation by analyzing execution paths and recovering hidden identifiers in the future. Finally, we only considered code written in Java, excluding access behaviors implemented in native or other languages, which may have led to false negatives. We will extend our static analysis to include native code in future work.

C. Ethical Consideration

We reported all findings to the relevant parties and assisted in resolving the identified issues. To date, all responding manufacturers have implemented the necessary fixes. To protect user privacy, we anonymized brand names mentioned in issue descriptions and omitted the specific names of vulnerable system properties and settings. This decision was made because many users continue to use outdated devices that no longer receive updates or choose not to apply them. As a result, vulnerable properties and settings containing non-resettable identifiers may still be present on user devices. Given that our findings are based on large-scale analysis of custom ROMs from various global vendors, publicly disclosing these findings could enable malicious actors to exploit them for unauthorized tracking. To mitigate this risk, we chose to anonymize specific findings and reduce the likelihood of privacy breaches.

VI. Related Work

A. User Identifiers and Tracking Practices.

Researchers have studied device identifiers and user tracking practices in mobile apps [2,10,47]–[49]. Razaghpanah et al. [3] conducted a global study on mobile tracking ecosystems. They identified 2,121 third-party advertising and tracking services, highlighting the heavy reliance on device identifiers for user tracking. Leith et al. [8] investigated data transmissions from mobile operating systems to vendors, revealing that device identifiers and other user data were sent to back-end servers even when users had configured their systems for minimal data sharing. Kollnig et al. [1] studied 24,000 Android and iOS apps to compare the performance of these operating systems in terms of user privacy, noting the widespread use of user identifiers in both. While these researchers examined the use of device identifiers for tracking by apps and the system, they did not explore the use of identifiers beyond those provided by the documented system APIs.

B. System Customization.

Numerous previous studies [19,22,23,50,51] have examined the security issues arising from the system customizations. Gamba et al. [6] conducted a large-scale study of pre-installed apps on Android devices. They uncovered relationships between supply chain actors and found that these apps often exhibit invasive or even malicious behaviors, including backdoor access to sensitive data. Lyons et al. [5] systematically investigated the sensitive information stored in Android system logs. They discovered that many device identifiers and user data were present in the logs, and that some privileged apps, including those from OEMs and large commercial companies, were collecting this information. El-Rewini et al. [21] conducted a large-scale study on residual APIs—unused custom APIs left in the system codebase. Analyzing 628 ROMs from seven vendors, they found these residual APIs are widespread and pose significant security risks, such as access control anomalies and potential exploitation. These studies have explored security issues introduced by system customizations, but none of them have involved the additional channels for obtaining device identifiers in custom systems.

Most relevant to our work, U2-I2 [15] conducted a systematic study on system-level protection of non-resettable identifiers. Through small-scale dynamic analysis, they revealed that many identifiers in custom Android systems, including those stored in system properties and settings, are not adequately protected, potentially leading to leakage. However, due to the limitations of dynamic methods, they are unable to perform large-scale analysis. In contrast, our static approach requires only system ROMs, making it far more efficient and scalable.

VII. Conclusion

In this paper, we present a comprehensive analysis of non-resettable identifiers in custom Android systems. We propose IDRadar, a systematic framework to identify sensitive and vulnerable system properties and settings. By applying our approach to 1,814 custom ROMs, we have identified thousands of vulnerable properties and settings, and further validation through remote real-device testing confirmed our findings. Our approach outperforms existing work in scalability, efficiency, and vulnerability coverage. We also investigated the root causes for such vulnerabilities, including overlooked system entries, complex access control rules, and covert supply chain involvement. In addition, we observed that the vulnerable implementations usually recur across devices from the same brand. Our study highlights the need for closer scrutiny of alternative access channels to non-resettable identifiers and improved safeguards for user privacy in system customizations.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (grant No.62572209) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

REFERENCES

[1] K. Kollnig, A. Shuba, R. Binns, M. Van Kleek, and N. Shadbolt, "Are iphones really better for privacy? a comparative study of ios and android apps," *Proceedings on Privacy Enhancing Technologies*, vol. 2022, no. 2, pp. 6–24, 2022.

[2] 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.

[3] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, P. Gill *et al.*, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *The 25th Annual Network and Distributed System Security Symposium (NDSS 2018)*, 2018.

[4] 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, 2020.

[5] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. Tapiador, S. Egelman, and N. Vallina-Rodríguez, "Log:{It's} big,{It's} heavy,{It's} filled with personal data! measuring the logging of sensitive information in the android ecosystem," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2115–2132.

[6] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1039–1055.

[7] C. Schindler, M. Atas, T. Strametz, J. Feiner, and R. Hofer, "Privacy leak identification in third-party android libraries," in *2022 seventh international conference on mobile and secure services (MobiSecServ)*. IEEE, 2022, pp. 1–6.

[8] D. J. Leith, "Mobile handset privacy: Measuring the data ios and android send to apple and google," in *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part II 17*. Springer, 2021, pp. 231–251.

[9] H. Liu, P. Patras, and D. J. Leith, "Android mobile os snooping by samsung, xiaomi, huawei and realme handsets," *techreport, Oct*, 2021.

[10] 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, pp. 603–620.

[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 *NDSS*, 2016.

[12] "Gdpr," https://gdpr-info.eu/, 2024.

[13] "California consumer privacy act," https://oag.ca.gov/privacy/ccpa, 2024.

[14] "Gdpr, art. 4 - definitions," https://gdpr-info.eu/art-4-gdpr/, 2024.

[15] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo, and J. S. Dong, "Post-gdpr threat hunting on android phones: Dissecting os-level safeguards of user-unresettable identifiers." in *NDSS*, 2023.

[16] H. Zhou, X. Luo, H. Wang, and H. Cai, "Uncovering intent based leak of sensitive data in android framework," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3239–3252.

[17] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio, "Trust, but verify: A longitudinal analysis of android oem compliance and customization," in *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021, pp. 87–102.

[18] Q. Hou, W. Diao, Y. Wang, X. Liu, S. Liu, L. Ying, S. Guo, Y. Li, M. Nie, and H. Duan, "Large-scale security measurements on the android firmware ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1257–1268.

[19] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "{FIRMSCOPE}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware," in *29th USENIX security symposium (USENIX Security 20)*, 2020, pp. 2379–2396.

[20] P. Liu, M. Fazzini, J. Grundy, and L. Li, "Do customized android frameworks keep pace with android?" in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 376–387.

[21] Z. El-Rewini and Y. Aafer, "Dissecting residual apis in custom android roms," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1598–1611.

[22] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 70–86.

[23] E. Blázquez, S. Pastrana, Á. Feal, J. Gamba, P. Kotzias, N. Vallina-Rodriguez, and J. Tapiador, "Trouble over-the-air: An analysis of fota apps in the android ecosystem," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1606–1622.

[24] "Artifact open source," https://github.com/security-pride/Custom-ROMs-Covert-Channels, 2024.

[25] 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 *Network and Distributed System Security Symposium (NDSS)*, vol. 10, 2018.

[26] 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, 2012, pp. 195–197.

[27] "System properties," https://source.android.com/docs/core/architecture/configuration#system-properties, 2024.

[28] "Security-enhanced linux in android," https://source.android.com/docs/security/features/selinux, 2024.

[29] "Mandatory access control," https://source.android.com/docs/security/features/selinux/concepts#mandatory_access_control, 2024.

[30] "Android open source project," https://source.android.com/, 2024.

[31] "strings(1) — linux manual page," https://man7.org/linux/man-pages/man1/strings.1.html, 2024.

[32] 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.

[33] "Android dumps," https://dumps.tadiphone.dev/dumps, 2024.

[34] "Oppo," https://www.oppo.com/en/, 2024.

[35] "Mediatek — powering the brands you love — incredible inside," https://www.mediatek.com/, 2024.

[36] "Xiaomi," https://www.mi.com/global/, 2024.

[37] "Lenovo," https://www.lenovo.com/, 2024.

[38] "Baidu," https://home.baidu.com/, 2024.

[39] "Alibaba," https://www.alibaba.com/, 2024.

[40] "Android_id," https://developer.android.com/reference/android/provider/Settings.Secure#ANDROID_ID, 2024.

[41] Z. Dong, T. Liu, J. Deng, L. Li, M. Yang, M. Wang, G. Xu, and G. Xu, "Exploring covert third-party identifiers through external storage in the android new era," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4535–4552.

[42] "Alibaba cloud," https://cn.aliyun.com/, 2024.

[43] "Mediatek — powering the brands you love — incredible inside," https://www.mediatek.com/, 2024.

[44] "U2-i2," https://uq-trust-lab.github.io/u2i2/, 2024.

[45] "Adding vendor-specific properties," https://source.android.com/docs/core/architecture/configuration/add-system-properties#add-vendor-properties, 2024.

[46] "https://support.google.com/googleplay/answer/1727131?hl=en," https://support.google.com/googleplay/answer/1727131?hl=en, 2024.

[47] E. Liu, S. Rao, S. Havron, G. Ho, S. Savage, G. M. Voelker, and D. McCoy, "No privacy among spies: Assessing the functionality and insecurity of consumer android spyware apps," *Proceedings on Privacy Enhancing Technologies*, 2023.

[48] N. Alfawzan, M. Christen, G. Spitale, N. Biller-Andorno *et al.*, "Privacy, data sharing, and data security policies of women's mhealth apps: scoping review and content analysis," *JMIR mHealth and uHealth*, vol. 10, no. 5, p. e33735, 2022.

[49] C. Han, I. Reyes, Á. Feal, J. Reardon, P. Wijesekera, N. Vallina-Rodriguez, A. Elazari, K. A. Bamberger, and S. Egelman, "The price is (not) right: Comparing privacy in free and paid apps," *Proceedings on Privacy Enhancing Technologies*, 2020.

[50] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1165–1178.

[51] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android {ROMs} via differential analysis," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 1153–1168.