Dialing Danger: Large-Scale Risk Assessment of Android Secret Codes in OEM Firmware

Ruoyan Lin*†, Shishuai Yang‡(✉), Fenghao Xu§, and Wenrui Diao*†(✉)
*School of Cyber Science and Technology, Shandong University
ruoyan@mail.sdu.edu.cn, diaowenrui@link.cuhk.edu.hk
†State Key Laboratory of Cryptography and Digital Economy Security, Shandong University
‡Zhengzhou University of Aeronautics, shishuai@zua.edu.cn §Southeast University, fhxu@seu.edu.cn

***Abstract*—Android's openness allows device manufacturers to deeply customize system functionalities, including the use of undocumented "secret codes", which are special dialer inputs that invoke privileged operations such as hardware testing, log recording, and configuration changes. Although these codes serve legitimate engineering purposes, their proprietary and non-standardized implementations across vendors result in an obscure and largely under-analyzed attack surface. Previous studies have examined only isolated cases and have not provided a comprehensive understanding of their prevalence, functionality, or security implications.**

In this paper, we present the first large-scale, cross-manufacturer analysis of secret codes embedded in system apps within Android firmware. We construct a dataset of 673 firmware images from 18 vendors and extract 218,354 APK files for analysis. By combining static analysis with LLM-assisted semantic interpretation, we identify and classify secret codes, analyze their functional categories, and assess the associated risks. Our results show that secret codes are widely deployed (averaging 103 per device), often redundant (20%-40%), and in some cases expose serious security vulnerabilities. Specifically, we discovered three real-world issues that allow attackers to bypass ADB authentication and gain privileged access. Furthermore, we observe a dual trend: the overall number of secret codes continues to grow, while some vendors have begun restricting access to sensitive functionalities. This indicates an uneven but increasing awareness of security concerns across manufacturers. To support future research, we also open-source our analysis tools and dataset.

***Index Terms*—Secret Code; Android; OEM Firmware; Security Analysis**

I. Introduction

The openness of the Android OS allows Original Equipment Manufacturers (OEMs) like `Samsung`, `Huawei`, and `OPPO` to deeply customize the system, creating unique features and interfaces while integrating seamlessly with their own hardware. In current research, the security issues caused by customized Android systems involve areas such as SELinux policies [41], custom permissions [27], and incompatible APIs [28]. One relatively common but lesser-known customization feature is the "secret codes" [15]. By entering a specific sequence of characters in the dialer interface, developers and technicians can access hidden diagnostic menus to perform functional tests or configure internal parameters. Although the Android Open Source Project (AOSP) [5] provides a standard method for handling these codes, manufacturers often extend it by adding proprietary, non-public code, and even using their own custom methods (such as proprietary broadcasts) to invoke these special functions.

The deep and non-standardized customization of secret codes by manufacturers has created a significant but under-researched attack surface in the Android ecosystem. These codes act as hidden "backdoors" capable of triggering powerful, high-privilege functions. For example, on `Honor` devices, entering `*#*#2846579#*#*` opens the *Background Settings* page, where users can configure USB port settings and AP log settings. Since these manufacturer-specific codes lack public documentation and vary significantly across different brands and even device models, their obscurity raises several critical security questions that have not been systematically assessed: ***How widespread is this practice? What high-risk functions are concealed behind these codes? Can attackers exploit these hidden entry points to bypass system security policies?***

The most relevant research includes vulnerability reports by Baptiste et al. [1] and Zhang et al. [2]. The former discovered the `EngineerMode` app on `OnePlus` devices, which was intended for factory testing but was not removed, allowing users to gain root access by entering a specific password. The latter used fuzzing to scan secret codes across major vendors, exposing flaws like log leakage and unauthorized resets that undermined Android's permission model. While these studies highlight the potential risks posed by secret codes, they remain limited in both scope and scale. Most focus on individual vendors or specific devices, without offering a broader analysis of secret code usage across the Android ecosystem. Specifically, no prior work has systematically examined how different manufacturers implement and manage secret codes, or how their usage has evolved over time.

Our Work. To address these unanswered questions, this work presents the first large-scale, cross-manufacturer investigation into the ecosystem of secret codes in system apps within Android firmware. We constructed a comprehensive dataset and employed a combined approach of static analysis and Large Language Model (LLM) to systematically extract these codes, categorize their functionalities, and assess their potential risks. Our study focuses on the functionalities provided by these codes, their redundancy and associated security risks, as well as the evolution of their usage over time, guided by the following research questions:

⇒ **RQ1:** *What functions do secret codes provide in mobile phones?*

⇒ **RQ2:** *Are there any secret codes that are redundant or pose security risks?*

⇒ **RQ3:** *What has been the trend of secret codes over the years?*

Our findings show that secret codes are a widespread practice, with devices containing an average of 103 codes, primarily for diagnostic and configuration purposes (Answering RQ1). This practice introduces code redundancy and security risks: redundancy rates frequently range from 20% to 40%, and more critically, we discovered and verified three exploitable vulnerabilities that bypass core security mechanisms, such as ADB authentication (Answering RQ2). Finally, we identify a dual trend over recent years: on one hand, the total number of secret codes has continued to grow; on the other hand, some manufacturers have begun enforcing stricter controls on high-risk codes. This indicates a growing awareness of security across the industry, though actions among manufacturers remain uncoordinated (Answering RQ3).

Contributions. The main contributions of this paper are:

- **New understanding.** We focus on manufacturer customization of secret codes on devices, an area that has been largely overlooked in prior research. Our study offers a comprehensive analysis of the use of secret codes across different manufacturers and systematically evaluates the resulting security and compliance implications.
- **Real-world measurement.** We conducted a large-scale analysis of Android devices by building a dataset of 673 firmware images from 18 manufacturers, covering 218,354 APKs. This provides a comprehensive view of how secret codes are used across different vendors.
- **Concrete attacks.** Based on the results of our analysis, we evaluated the security implications of secret codes on seven real-world mobile devices and found 142 secret codes with security risks. Moreover, 3 of these devices are vulnerable to attacks that bypass ADB debugging permission configurations.

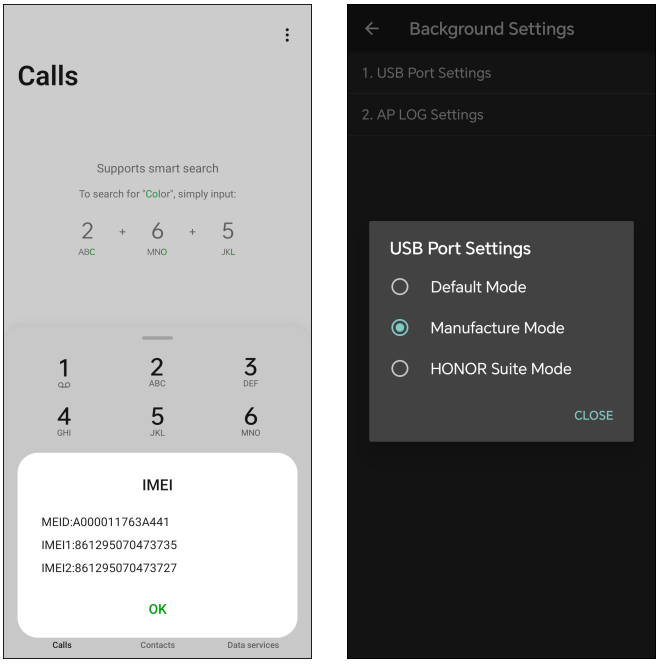
Responsible Disclosure. Following the responsible disclosure policy, we reported the identified vulnerabilities to the corresponding vendors. Honor acknowledged our report and has released a patch to fix the vulnerability. In addition, the vendor assigned CVE-2025-57840 and awarded a bounty for our report.

II. Background

In this section, we introduce the necessary background for this work. We first describe the meaning of Android secret codes and the mechanism by which the Android OS handles them. Second, we discuss the differences between the AOSP [5] and device manufacturer customization ecosystems.

A. Secret Code Mechanism

"Secret Codes", sometimes referred to as "Debug Codes" or "Hidden Codes" [15], are specially formatted character sequences embedded in system apps within the Android OS. These codes are designed by device manufacturers, and developers typically enter these codes through the system's dialer app to access hidden test menus, diagnostic tools, or internal features. As illustrated in Figure 1, when entering a specific formatted character sequence in the dialer app, such as `*#06#` shown in Figure 1 (a) and `*#*#2846579#*#*` in Figure 1 (b), the dialer app recognizes the pattern and does not attempt to make a call. In some cases, particularly those for simpler or internal tasks, the dialer handles the codes directly. These actions can range from displaying a simple dialog with system information to more covert operations, such as modifying system settings or starting a background service to perform time-consuming tasks. In such cases, the entire process can be imperceptible to the user, as it may not produce any visible changes to the user interface.



(a) `*#06#` (b) `*#*#2846579#*#*`

Fig. 1: Secret Codes Within Devices.

```
<receiver
  android:name="com.hihonor.android.
      projectmenu.ProjectMenuReceiver"
  android:permission="com.hihonor.permission
      .SECURITY_ACTIVITY"
  android:exported="true">
  <intent-filter>
    <action android:name="android.provider.
        Telephony.SECRET_CODE"/>
    <data android:scheme="
        android_secret_code"
          android:host="2846579"/>
  </intent-filter>
</receiver>
```

Listing 1: Example of a Secret Code Receiver.

In another case, the dialer app internally constructs a specific Intent object and broadcasts it to system components that may be interested in triggering the corresponding secret functionality. According to the AOSP definition, this Intent typically contains the following two key fields: (1) **Action:** `android.provider.Telephony.SECRET_CODE`, or its updated form `android.telephony.action.SECRET_CODE`. It serves as an explicit "signal" to the system indicating that this is an event related to a secret code. (2) **Data:** The secret code itself (e.g., `2846579`) is encapsulated in a URI object, usually formatted as `android_secret_code://2846579`. The receiver can extract the specific code entered by the user by parsing this URI.

Any app that intends to respond to a specific secret code must statically register a `BroadcastReceiver` [7] in its `AndroidManifest.xml` file, as illustrated in Listing 1. This receiver needs to declare an `<intent-filter>` that matches the above-mentioned `action` and `data` scheme. When the system sends out the broadcast, the matching `BroadcastReceiver` will be triggered, and its `onReceive` method will be invoked to execute the specific logic associated with the secret code.

B. AOSP and Manufacturer Customization

The key to understanding the ecosystem of secret codes lies in understanding the openness and fragmentation of the Android OS. AOSP is the open-source base version of the Android OS maintained by Google. It includes the core functionalities of Android and a set of standard secret codes, primarily used for general debugging and testing (such as the `*#06#` mentioned earlier). It is this open-source nature that enables device manufacturers to build upon and diverge from the AOSP baseline. To achieve market differentiation, OEMs such as `Samsung`, `Huawei`, `Xiaomi`, and `Vivo` undertake substantial secondary development and deep customization. The principal drivers for this are to establish a distinct brand identity, to ensure the integration and optimization of proprietary hardware, and to adhere to the market and regulatory requirements of specific locales. While this customization is a source of the Android ecosystem's diversity, it also poses a potential security risk.

This deep customization strategy also extends to the realm of secret codes. Manufacturers not only retain most of the standard AOSP codes but also add a large number of private, non-public secret codes for their internal research and development, testing, and after-sales processes. These codes serve as "backdoors" to access proprietary features, such as performing hardware diagnostics and calibrations on specific hardware components (e.g., particular camera or fingerprint scanner models). To manage these private codes and avoid conflicts with standard mechanisms or other apps, manufacturers typically define their own private broadcast actions to handle them. For example, instead of using the standard broadcast action defined by AOSP, `Vivo` defines and uses its own action (`android.provider.Telephony.VIVO_SECRET_CODE`) to receive and handle secret codes specific to its brand's devices. This non-standardized practice leads to a situation in which each brand's devices may have unique sets of secret codes and response mechanisms, distinct from those used by devices from other brands.

C. Threat Model

In our threat model, the attacker exploits a situation where the device is already unlocked and left unattended. For instance, the user might briefly leave the phone on a desk without manually locking it. During this window, the attacker can launch the dialer app and enter a secret code to trigger sensitive system functions.

III. Methodology and Dataset

As illustrated in Figure 2, the methodology adopted in this study consists of three core phases, each designed to progressively build upon the previous one to enable comprehensive discovery and evaluation of secret codes in Android firmware:

- *Dataset Construction and Preprocessing.* We begin by collecting and filtering a diverse set of firmware images from multiple vendors, followed by extracting embedded system apps to serve as the analysis foundation.
- *Secret Code Identification and Extraction.* We then define the structural patterns of secret codes and apply two complementary static analysis techniques to extract them from firmware apps, along with their execution entry points.
- *Functional Analysis and Risk Assessment.* Finally, we perform static call path analysis and leverage LLMs to categorize the functionality of each code and assess its potential security risks and necessity.

A. Dataset Construction and Preprocessing

Firmware Collection. First, we constructed a firmware dataset to extract and evaluate the security of secret codes. The firmware samples used in this study are primarily sourced from `Android Dumps` [4], a well-known repository that provides publicly available Android firmware images for the research community. Since the number of firmware images varies across brands in Android Dumps, we also downloaded additional firmware from the official websites of brands with fewer samples [9], [10], [11], [12].

Dataset Cleaning and Filtering. The `Android Dumps` dataset includes a wide range of firmware versions, including official releases and beta builds for various devices, such as phones, TVs, and smartwatches. To ensure our analysis remained both accurate and relevant, we applied a thorough data cleaning process. For this study, we focused exclusively on official release firmware intended for mobile phones. Specifically, our cleaning process involved the following four steps:

- *Duplicate Firmware Filtering.* We first calculated the MD5 hash of each firmware image file and removed any duplicates to avoid redundant analysis of the same firmware.
- *Device Scope and Integrity Filtering.* In order to ensure the integrity and relevance of our dataset, we first filtered out any firmware package lacking a `build.prop` file, as it is essential for identifying critical metadata like device model and version. We then further refined the dataset by parsing properties within the

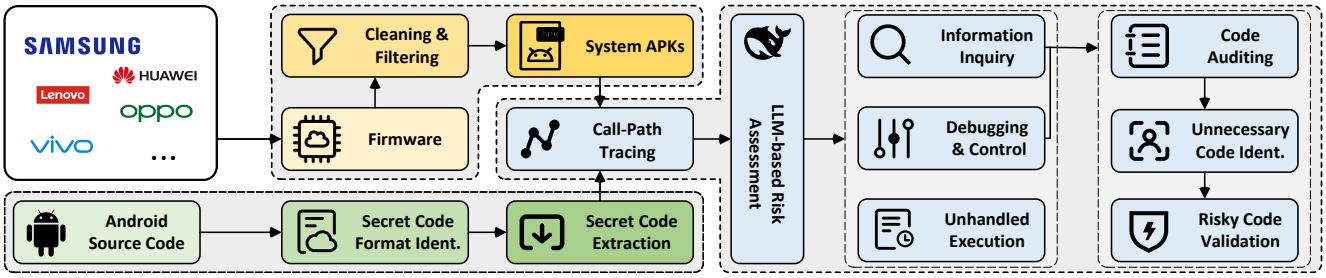


Fig. 2: Overall Analysis Process.

build.prop file, such as `ro.product.system.device` and `ro.product.system.model`, to remove firmware for non-phone devices (e.g., TVs, watches) and non-production builds (e.g., developer testing, debugging builds).

- *Release Type Filtering.* To retain only the final products intended for market release, we filter the firmware based on two properties in the `build.prop` file. We check for "user" in the `ro.build.type` [14] property and "release-keys" in `ro.build.tags` [13], as these respectively indicate a user-facing build type and the use of a release signing key.
- *Timeliness Filtering.* To ensure the analysis reflects current trends in secret codes, we further filtered our dataset to include only firmware built after 2020, using the build date from `ro.build.date.utc` property in `build.prop` file.

After the above cleaning and filtering process, we ultimately retained a dataset consisting of 673 firmware images from 18 different brands for analysis. We extracted the system apps from each image as a basis for subsequent analysis phases, including secret code extraction, behavioral analysis, and security evaluation.

B. Secret Code Identification and Extraction

Based on the constructed firmware dataset, our core task is to comprehensively extract secret codes for security evaluation. We first need to accurately identify the format of secret codes in the source code, such as whether they consist purely of digits or a combination of digits and `*` or `#`. Based on these identified formats, we then develop automated scripts to batch extract all relevant secret codes from the dataset.

Secret Code Format Identification. We identify the format of secret codes by distinguishing between standardized codes defined by AOSP and proprietary codes introduced by device manufacturers: (1) *Secret Codes defined by AOSP.* The code formats are listed in the AOSP source file `SpecialCharSequenceMgr.java` [16], where the comments of `handleSecretCode` method explicitly define two formats: `*#*#<code>#*#*` and `*#<code_starting_with_number>#`. (2) *Secret Codes defined by Manufacturers.* In the process of vendor customization, secret codes beyond those defined by AOSP are sometimes used, such as `##467#`. To identify potential manufacturer-defined secret code strings, we extract all strings composed solely of the characters `*`, `#`, and digits from the firmware, as these are the only characters used in dialer input. We then manually verify each extracted string to determine whether it is a secret code.

Some secret codes, when dispatched via an Intent, typically specify a particular action. This action is key to identifying the component that processes the code. Based on their origin, these actions can also be categorized into two types: (1) *Actions Defined by AOSP.* As specified in the AOSP source file `TelephonyManager.java` [17], the comment for the `ACTION_SECRET_CODE` field indicates that the broadcast action for handling secret codes should be either `android.telephony.action.SECRET_CODE` or its legacy version, `android.provider.Telephony.SECRET_CODE`. (2) *Actions Defined by Manufacturer.* In addition to defining custom secret code formats, device manufacturers also introduce non-standard Intent actions to receive and process these codes. To systematically identify such custom actions, we scanned the `AndroidManifest.xml` files of all apps in the firmware and applied the following identification rules: (1) First, we retrieved all values of the `name` attribute within `<action>` tags that contain the keyword `SECRET_CODE`. (2) Then, we excluded standard actions defined by AOSP from the results, with the remaining entries considered as manufacturer-defined actions.

Secret Code Extraction. To extract all secret codes from firmware, we statically analyze system apps by reviewing their `AndroidManifest.xml` files and hardcoded strings. This works because dialed codes are either handled directly by the dialer app or broadcast to another app for processing:

- *Manifest-based extraction.* To respond to secret codes, an app must statically declare a `BroadcastReceiver` in its `AndroidManifest.xml` file to listen for the corresponding action. We analyze each app's declared receiver components and check whether any `<action>` element within their `<intent-filter>` corresponds to a `SECRET_CODE` action. In the `<intent-filter>`, its subtag `<data>`'s `host` attribute specifies the secret code that the `BroadcastReceiver` can handle. Therefore, if such an action is found, we further extract the `host` value from the `<data>` element under the same `<intent-filter>`.
- *Pattern-based extraction.* If an app does not specify the `BroadcastReceiver` that handles a secret code through the `<data>`, it can only hardcode the logic in the code, triggering different actions based on the secret codes. Therefore, we can extract hardcoded strings that match

these patterns directly by string matching, based on the formats of secret codes previously summarized from apps. However, not all strings from pattern-based extraction are actual secret codes. Some may simply be regular strings that happen to fit the format. The key distinction is that secret codes are typically compared in the code to trigger specific functionality. Such comparisons are commonly implemented using conditional statements such as `if` or `switch`, or through the use of the `equals` method in smali code. Ordinary strings, however, are not involved in such comparisons. Therefore, based on the previously identified secret code patterns, we first construct high-precision regular expressions to extract secret codes and then check whether a candidate string matching the format is involved in an equal comparison. If not, it is considered an ordinary string and filtered out. These expressions are applied to all code files to specifically identify hard-coded strings used in conditional comparisons, enabling the accurate extraction of secret codes.

C. Functionality Analysis and Risk Assessment

Call Path Tracing and Analysis. To determine the final operation triggered by the secret code and assess its security risks, we need to trace the execution path of the code. Our secret code extraction process provides an ideal starting point for functional analysis, as each method not only extracts the secret code but also identifies its corresponding entry point within the app. Specifically: (1) For codes identified via Manifest-based extraction, the entry point is the `onReceive` method of the corresponding `BroadcastReceiver`. (2) For codes found through Pattern-based extraction, the entry point is the method containing the hardcoded string.

Then, we use `AndroGuard` [3] to construct the app's call graph. Starting from the analysis entry point, we perform control-flow analysis along the call graph to determine the execution path. When the path involves an Intent dispatch (i.e., intra- or inter-app communication), we conduct value analysis to resolve the Intent object and its parameters (such as target *package name*, *Activity*, and *Action*). This enables the accurate tracing of cross-component and cross-app calls, continuing until one of the following three behavioral endpoints is located: (1) a new UI is displayed (e.g., an *Activity* is launched); (2) a popup message is shown (e.g., a *Toast* or *Dialog*); or (3) there is no UI change (i.e., the code executes in the background with no UI feedback).

LLM-based Risk Assessment. After identifying the final behavior triggered by each secret code, we leverage the `DeepSeek-R1` [8] to perform an initial automated assessment of its functionality. Based on the results from the previous step, we extract contextual information and code snippets for each secret code to serve as part of the input to the LLM: (1) *UI Display.* Extract all visible text elements from the target UI component (e.g., *Activity* or *Fragment*). (2) *Popup Message.* Extract the text content passed to the `Toast` or `Dialog`. (3) *No UI Change.* Extract the code snippets and their strings from the code path starting at the entry point and ending at the final activity it jumps to. Considering the potential hallucination issues in LLMs, we use a voting mechanism during LLM-based reasoning to improve the reliability of the results.

Functional Categorization. To classify and evaluate secret codes' functionality, we design an LLM task prompt for a two-level classification based on the types of secret codes identified through manual analysis. We first perform a primary classification of secret codes into *Information Inquiry* (functions related to read operations), *Debugging and Control* (functions related to write and execute operations), or *Unhandled Execution* (no function is available due to unresolved Activity/Service/Broadcast Intent). Then, we conduct a secondary classification to further subdivide the function within the first three primary categories and assess its risk level.

(1) **Information Inquiry.** This category of secret code is used to retrieve hardware or software information from the device. Based on the relevance of the retrieved information to the user or the device, the codes are classified into three risk levels:

- *Unique Device Identifier Codes.* Data such as IMEI and IMSI [6], can uniquely identify a user or device and enable tracking across apps and platforms. Even if the user clears app data or resets the device, the information may still be used to re-establish identity, posing a high security risk.
- *Potential Threat Codes.* While data such as software and firmware versions is not uniquely identifying in isolation, it can pose a significant risk when aggregated. For instance, attackers can leverage version numbers to infer the presence of unpatched vulnerabilities and subsequently launch targeted attacks. Thus, the combination of multiple such individual attributes can enable software fingerprinting or facilitate the exploitation of known vulnerabilities, constituting a moderate security risk.
- *General Device Status Codes.* This type of data, which includes common information like screen parameters and battery status, is used primarily for device adaptation or status detection and has only a weak connection to user identity or system security. Although it could potentially be used for fingerprinting and tracking through excessive aggregation, these attacks are typically complex and offer limited benefits in normal use cases. Therefore, it is considered a low security risk.

(2) **Debugging and Control.** This category covers secret codes used for sending control commands, device diagnostics, and configuration changes. Based on their functionality, they can be further divided into the following three categories:

- *Configuration Operation Codes.* These codes perform write or reset operations, affecting both app-level and system-level configurations. Examples include enabling ADB over USB or triggering a factory reset. Since these codes can change system states and user data, they are prone to causing misconfigurations or data loss. Due to their potential impact on user privacy and system integrity, they present a higher security risk.
- *USSD Codes.* These codes initiate Unstructured Supplementary Service Data (USSD) network requests [18] used

for purposes such as setting up call forwarding. If an attacker uses USSD codes for call forwarding to redirect all incoming calls to their own number, the user will not receive other calls. Furthermore, the attacker could also take over accounts, reset passwords, and ultimately hijack funds or services. Therefore, they pose a high security risk.

- *Diagnostic Test Codes.* These codes are used to trigger diagnostic tests and help users identify basic issues. For example, they can be used to check whether components such as the microphone or camera are functioning properly. However, if the operator lacks certain specialized knowledge, improper operation can still cause abnormal behavior. Since their working mechanism is limited to activating test modules and does not modify system settings, user data, or firmware, they do not endanger the device's core functionality or data integrity, resulting in a low overall security risk.

(3) **Unhandled Execution.** Secret codes in this category fail due to interrupted inter-app transition. When an app (caller) sends an Intent to another app's component, if the target app is unavailable on some devices, the secret code's request goes unanswered, making it impossible to determine or analyze its intended function. Since the secret code request fails to execute the function, it does not affect user data, system settings, or device state, and also poses no direct security risks such as data leakage, configuration corruption, or device failure. The potential harm to the device is minimal. Therefore, unhandled execution codes are considered low risk.

Secret Code Auditing. To assess the potential security risks of secret codes, we manually verified those previously flagged as medium- and high-risk on seven real-world devices from 6 different brands, released between 2020 and 2025. We adopted this focused strategy because low-risk secret codes generally pose a minor actual threat, as they typically involve high exploitation costs, offer low rewards, or have restricted functionality. By concentrating on the most critical areas, we aim to accurately identify every risk point that poses a substantial threat and analyze it in depth.

In addition, to enhance the security and privacy of the device, we use the LLM to assess the redundancy of secret codes, excluding those categorized as "Unhandled Execution". Specifically, in official consumer-facing devices, the primary users are the device owners, and after-sales services may need to access the device in case of malfunctions. Therefore, if any functionality of a secret code is not useful to the user or for after-sales services, a redundancy problem exists.

IV. Measurement and Findings

This section summarizes the key findings from our empirical research, which addresses the research questions in Section I.

RQ1. What functions do secret codes provide in mobile phones?

Distribution Overview. We systematically evaluated the secret codes in 673 firmware samples from 18 brands, totaling 218,354 APKs. The results are shown in Table I. Due to the uneven distribution of firmware images from different manufacturers in the dataset, the number of firmware samples we collected for each brand also shows a similar imbalance, with `Samsung`, `Motorola`, and `Vivo` having the largest number of samples. Overall, 5.58% of the apps contain secret codes, with an average of 103 secret codes per firmware. This indicates that the secret codes in each firmware are not centrally managed by a single Dialer app or EngineerMode app but are instead distributed across multiple apps, which increases the difficulty of security auditing. In addition, although only 5.58% of system apps in every firmware include secret codes, these codes are widely distributed across different firmware samples, suggesting that many devices integrate secret code functionality as part of their system-level design.

As shown in Table I, however, the number and distribution of these codes vary significantly from brand to brand. Notably, `Samsung` stands out with the highest proportion of apps containing secret codes (7.47%) and has the highest average number of secret codes per firmware (249), significantly exceeding other brands. ***The main reason is that all Samsung firmware includes the*** `DeviceKeystring` ***app (***`com.sec.android.app.factorykeystring`***), which contains a series of secret codes to implement a full suite of factory testing and diagnostic functions.*** Its functionalities cover device information queries (e.g., serial number, manufacturing origin, CPU version, battery status), support for firmware version checks, OTP verification, and failure history access. It also provides various sensor and audio tests, along with operations such as S-Pen testing, vibration testing, firmware updates, and UART enabling. Although the exact number of secret codes within this app varies slightly between firmware versions, it contains at least 74 and up to 94 codes. The number of secret codes in this single app alone exceeds the average number of codes per firmware for all brands except `Tecno`.

Function Categories. Based on the detection results, ***the primary function of secret codes in most vendor devices is diagnostic testing.*** These codes allow testing of various components, including the SIM, key, touchpanel, display color, speaker, microphone, GPS, Wi-Fi, and charging. They can be used to assist in detecting device status during the development stage, daily use, or maintenance processes, ensuring both software and hardware features are operating correctly. The general device status codes under the Information Inquiry category also serve a similar role. These codes display basic device information, such as battery status, light sensor lux value, LCD type, RTC time, and SAR values, which help verify whether relevant functions are working as expected.

Configuration operation codes also make up a significant portion of the firmware across most manufacturers and serve various functions. They can open feedback pages, allowing users to record screen activity when issues occur and send the recordings back to the manufacturer. Other codes support logging features such as `ModemLog`, and `NetworkLog`. Some codes enable debugging features for individual apps, such as activating tracedebug, 3A async debug, or color dump in the

TABLE I: Distribution and Characteristics of Secret Codes.

#	Vendor	Firmware	APKs	SCs	APKs with SCs	SCs / F	Information Inquiry			Debugging and Control			Unhandled Execution
							UDI (H)	PT (M)	GDS (L)	CO (H)	USSD (H)	DT (L)	
1	Google	22	3876	197	70 (1.80%)	9	27	0	4	102	10	23	31 (15.74%)
2	Honor	11	3265	396	100 (3.06%)	36	40	6	28	39	73	188	22 (5.56%)
3	iQOO	8	2493	421	105 (4.21%)	53	71	9	30	85	17	170	39 (9.26%)
4	Lenovo	7	1563	205	75 (4.80%)	29	14	14	31	67	4	64	11 (5.37%)
5	Meizu	28	6415	803	267 (4.16%)	29	74	33	58	170	52	321	95 (11.83%)
6	Motorola	110	38757	6147	1852 (4.78%)	56	289	89	443	3136	153	1646	391 (6.36%)
7	Nokia	57	13747	1824	727 (5.29%)	32	156	61	88	522	67	686	244 (13.38%)
8	Nubia	11	5132	439	97 (1.89%)	40	30	3	33	100	74	172	27 (6.15%)
9	OnePlus	34	10693	1838	518 (4.84%)	54	93	52	87	775	75	591	165 (8.98%)
10	OPPO	13	4483	935	180 (4.02%)	72	54	51	38	438	35	289	30 (3.21%)
11	Panasonic	3	885	56	20 (2.26%)	19	6	0	6	9	6	23	6 (10.71%)
12	Realme	22	7039	1609	333 (4.73%)	73	80	95	76	701	66	554	37 (2.30%)
13	Redmi	28	7646	1958	527 (6.89%)	70	86	19	153	387	133	716	464 (23.70%)
14	Samsung	171	67579	42617	5046 (7.47%)	249	1433	1550	3225	8944	2031	12744	12690 (29.78%)
15	Tecno	19	5241	1544	215 (4.10%)	81	40	40	29	528	69	501	337 (21.83%)
16	Vivo	88	27120	5945	1454 (5.36%)	68	490	80	528	1391	240	2805	411 (6.91%)
17	Xiaomi	27	7963	1672	463 (5.81%)	62	82	32	143	316	112	802	185 (11.06%)
18	ZTE	14	4457	589	132 (2.96%)	42	42	3	38	178	113	202	13 (2.21%)
	Total	673	218354	69195	12181 (5.58%)	103	3107	2137	5038	17888	3330	22497	15198 (21.96%)

Note: SCs: secret codes, APKs with SCs: number of apps containing secret codes, SCs/F: secret codes / firmware, UDI: unique device identifier codes, PT: potential threat codes, GDS: general device status codes, CO: configuration operation codes, USSD: USSD codes, DT: diagnostic test codes, H: High Risk, M: Medium Risk, L: Low Risk.

camera app. Additional functions include enabling the device's debugging port or restoring the device to factory settings.

For unhandled executions, we analyze the reasons why a secret code cannot successfully trigger its intended behavior. There are two causes: (1) *Invalid Component Declaration.* A non-standard component name declaration prevents the system from correctly resolving the component and, therefore, causes the secret code's execution to fail. For example, Google's `CalendarProvider` app (`com.android.providers.calendar`) declares a component named `CalendarDebugReceiver`. Since the declaration for this receiver uses neither a fully qualified name nor a relative package name (e.g., `.CalendarDebugReceiver`), the system cannot locate the component, resulting in an execution failure. (2) *Target Component Missing.* This includes two scenarios. The first is an internal component inconsistency, in which an app's manifest file declares a receiver, but the corresponding code implementation is missing from the app. The second is a cross-app dependency failure, where an app needs to start another target app via an Intent to complete the process, but that target app is not installed on the device. For example, Xiaomi's `Catchlog` app (`com.bsp.catchlog`) declares a `.CatBroadcastReceiver` component, but the concrete implementation of this component does not exist within the app.

Answers to RQ1

Secret codes mainly support diagnostic testing and configuration operation tasks, such as logging and debugging. They are widely present across firmware, especially in apps from major vendors like Samsung. However, some codes fail due to invalid declarations or missing components.

RQ2. Are there any secret codes that are redundant or pose security risks?

Redundant Secret Codes. By analyzing the complete functionalities of each secret code, we found that many codes contain redundant operations. We calculated the percentage of redundant secret codes for each vendor. For most vendors, this ratio ranges between 20% and 40%. The percentages of four vendors, `Realme` (57.99%), `OPPO` (54.65%), `Tecno` (44.75%), and `Xiaomi` (42.52%), exceed 40%. In contrast, `Google` (17.26%) and Panasonic (3.57%) have fewer redundant codes, both below 20%. This indicates that the redundancy is not an isolated issue in the industry but a fairly prevalent problem. Moreover, the usage of redundant secret codes varies significantly across manufacturers.

Leaving redundant secret codes in officially released smartphones introduces several risks. These codes are primarily designed for developers and internal testing (e.g., configuring system properties, managing logs), which do not align with the actual needs of users. Users' daily operations and after-sales maintenance are typically handled through secure, stable graphical interfaces, making these residual background functions unnecessary in the final product. This functional redundancy not only leads to software bloat and expands the device's attack surface, but can also introduce potential security risks if these codes are misused to modify system behavior, ultimately increasing the device's long-term maintenance costs.

Risky Secret Codes. Based on the methodology proposed in Section III-B, we identified secret codes on seven real devices from six different brands, including `Honor`, `Huawei`, `iQOO`, `OPPO`, `Redmi`, and `Vivo`. We then used the LLM to assess the security risk levels of these secret codes and manually verified 142 codes classified as medium and high risk. To ensure the

reliability of our findings, each code was evaluated twice. Although some codes cannot be activated on real devices, this comprehensive audit still revealed three critical and exploitable vulnerabilities related to USB debugging.

Normally, enabling ADB debugging requires a multi-step process: tapping the `Build number` seven times, enabling the developer options page, turning on USB debugging, and finally confirming the RSA key. On certain devices, password authentication is also required during this process to restrict ADB access and protect the device. However, by entering certain secret codes, we were able to bypass the password check on Huawei and Honor devices or the RSA key confirmation on the Vivo device, effectively circumventing the protection mechanisms enforced by the security model.

Similarly, when an Honor device is connected by USB for file transfer, a password is required for secondary confirmation. Using the secret code, we can also bypass this verification and directly enable the corresponding function.

We reported the three discovered vulnerabilities to the corresponding vendors. Honor confirmed our findings, while Huawei and Vivo responded with explanations indicating that there were no security concerns. Huawei stated that such functionality needs the device to have been authorized for USB debugging, and the authorization information must not have been cleared. The current functionality is consistent with the design specification. Likewise, Vivo explained that this feature requires actively entering the secret code and enabling the "Debugging Port" option, and that the UI label "Debugging port" already indicates its function.

Answers to RQ2

Redundant secret codes are commonly found in devices from various brands, resulting in software bloat and an increased attack surface. Risky secret codes also exist and can be exploited to bypass ADB security measures, posing serious security threats by enabling unauthorized access and control.

RQ3. What has been the trend of secret codes over the years?

We analyzed the annual average number of secret codes in firmware in the dataset from 2020 to 2025: 93, 94, 92, 110, 135, and 132, respectively, indicating an overall upward trend. The increase is mainly caused by two factors: on one hand, this might be due to manufacturers reserving more entry points for testing, after-sales maintenance, and debugging during feature integration; on the other hand, it reflects that some manufacturers tend to retain engineering interfaces when balancing security and maintainability, aiming to reduce support costs and improve development efficiency.

This upward trend reflects a deeper issue: ***although the industry has made progress in general security measures, considerations of development convenience seem to take precedence over the security principle of minimizing attack surfaces in final products when handling engineering interfaces such as secret codes.*** However, this general trend of prioritizing convenience does not mean the issue is completely ignored. Our analysis reveals that some manufacturers, in particular, are aware of the existence of high-risk secret codes and have begun to take targeted measures to control them. For example, during our manual analysis of secret codes, we observed that the activation conditions for the same secret code can vary across versions of the same app. We will illustrate this point using the secret code `*#8011#` on two OPPO devices: the `A32` released in 2020 and the `A3i Plus` released in 2025.

On `OPPO A32`, enabling the ADB mode typically requires a series of steps: navigating to `Settings` - "`About Phone`" - "`Version`", tapping the `Build number` seven times, entering the lock screen password, and then accessing developer options to enable USB debugging. This process enforces user verification before allowing debugging access. However, when entering `*#8011#` on the `A32`, a pop-up message appears stating: *"this command only for debug, please reset adb switch in development options to fix it."* In this case, the `OppoEngineerMode` app (`com.oppo.engineermode`), which handles the secret code, first calls the `UsbConfigManager (context).enableAdbFunction(true)` method, and then uses the `Settings.Global.putInt` method to set the value of `adb_enabled` to `1`, enabling ADB debugging.

In contrast, on `OPPO A3i Plus`, entering the same secret code no longer enables the ADB function. During the reverse engineering of the APK, we found that although the secret code cannot be used to execute the same function, the code to set `adb_enabled` to `1` still exists within the handling logic of `*#8011#` in the `OplusCommercialEngineerMode` app (`com.oplus.engineermode`, a newer version of `OppoEngineerMode`). Before this method is called, a whitelist filtering is applied. Only the specified secret codes can execute their functions, preventing the other secret codes from being processed. As a result, it is no longer possible to enable ADB debugging using this secret code.

Answers to RQ3

Secret codes have increased over the years due to development needs, but some manufacturers are now applying controls to reduce risks. This shows growing awareness and improving management of secret codes in real devices.

V. Case Studies

This section presents three real-world case studies, each illustrating a distinct category of problematic secret codes discovered in our analysis: unhandled, unnecessary, and risky.

A. Case Study 1: Unhandled Secret Codes

When we enter the secret code `*#1111#` in the dialer app on the `Vivo Y300`, the system sends a broadcast and uses the `setPackage` method to specify the `DemoVideo` app (`com.vivo.demovideo`) to handle this secret code. However, the `DemoVideo` app does not actually exist on the `Vivo Y300` device. As a result, there is no app on the device capable of

receiving this broadcast, causing the secret code's processing flow to be interrupted, making it an unhandled secret code.

Impact. Such unhandled secret codes are remnants of deprecated or incomplete functionality in the system. It not only reflects poor maintenance or oversight in firmware customization but also exposes the system to potential security risks. From a security perspective, this misconfiguration creates a clear attack vector. An attacker could craft a malicious app with the exact same package name (i.e., `com.vivo.demovideo`) and trick the user into installing it. Once installed, the malicious app could intercept broadcasts intended for internal system components, and executing unauthorized actions could lead to data leakage or privilege escalation.

B. Case Study 2: Unnecessary Secret Codes

When the secret code `*#*#7562#*#*` is entered on an `OPPO A3i Plus` device, the system triggers the launch of the `CalendarDeveloperModeActivity` component within the `Calendar` app (`com.coloros.calendar`). This activity serves as a "Calendar Developer Page," offering features such as bulk importing 1,000 schedule entries, switching cloud services and account environments to testing or development modes, and resetting the version number of public holiday data. However, users typically use the `Calendar` app to view, create, and edit events, meaning the functionality triggered by this secret code does not align with regular user interactions with the app. Moreover, even if the device or app encounters issues and is sent for repair, these developer features do not assist in restoring the device. As a result, despite the presence of these developer features in officially released devices, they have no practical use case and are considered unnecessary secret codes.

Impact. The internal testing tools on this page, such as bulk data import and environment switching, are not designed for end users and pose multiple security risks. First, they threaten the integrity of user data, as debugging operations like "bulk import" or "resetting versions", if misused, lead to data corruption or deletion. Second, attackers can exploit this by using social engineering strategies to deceive users into entering the code, enabling them to remotely damage data or configure the device maliciously. Furthermore, exposing developer features in consumer-facing products directly violates the core security principle of *least privilege*, unnecessarily expanding the attack surface and weakening the overall security of the product.

C. Case Study 3: Risky Secret Codes

On `Honor Play 9T Pro` and `Honor Magic 6` devices, under normal circumstances, when the device is connected to a computer via USB, the user must enter the lock screen password to authenticate before enabling developer mode and ADB debugging. However, as shown in Figure 1b, by entering `*#*#2846579#*#*` in the dialer app, the engineering menu in `ProjectMenu` (`com.hihonor.android.projectmenu`) app can be accessed. Within the engineering menu, under `Background Settings - USB Port Settings`, selecting `Manufacture Mode` allows ADB debugging to be activated without requiring lock screen password authentication.

Likewise, users are typically required to authenticate by entering the lock screen password on the device before they can change the USB mode from "Charge only" to "Transfer files". By entering the secret code `*#*#2846579#*#*` to open the engineering menu in `ProjectMenu` app, and selecting `HiSuite Mode` under `Background Settings - USB Port Settings`, the device can switch from "Charge only" to "Transfer files" mode without lock screen password verification.

Impact. This vulnerability poses serious security risks. An attacker who gains physical access to the device or tricks the user into entering a specific secret code can bypass critical system security policies. It could allow the attacker to establish an ADB connection or enable USB-based file transfer mode without requiring any authentication from the user. Once these features are enabled, the attacker can perform malicious activities such as unauthorized command injection or the leakage of sensitive data. These actions may not only compromise the device's confidentiality and integrity but also enable further exploitation. `Honor`'s security team has confirmed the vulnerability and awarded a bounty for it.

VI. Discussion

In this section, we propose mitigation measures and present the threats to the validity of our study.

A. Mitigation Measures

Based on our research findings, we propose the following mitigation measures to address the security risks posed by secret codes in production devices:

Strict Code Management for Production Builds. The most effective mitigation strategy is to remove unnecessary secret codes from consumer-facing firmware. Manufacturers should leverage build configurations (e.g., distinguishing between `user` and `userdebug/eng` builds) to separate from internal debugging features from official release versions. Developer menus, diagnostic tools, and other engineering-only features should be stripped from production builds during the final packaging stage. This approach aligns with the *least privilege* principle and fundamentally reduces the attack surface.

Additionally, to mitigate the risks caused by flawed secret code implementations, manufacturers should verify the correctness of `BroadcastReceiver` declarations to prevent invalid component formats and validate the Intent dispatch logic to ensure that any target package invoked by a secret code (as demonstrated in our detailed `Vivo` case) is present in the final build, preventing unhandled secret codes in production.

Enforce Access Control on Retained Codes. If certain secret codes must be retained, such as those required for specific user-facing features or after-sales support, their execution logic should be protected by the security mechanisms. For example, high-privilege operations like enabling ADB or modifying system configurations should always follow the standard user authentication process. These sensitive actions must only be executed after the system has successfully verified the user's identity through established and reliable methods, such as a lock screen password, PIN code, or biometric authentication.

B. Threats to Validity

Internal Validity. Due to custom implementations by manufacturers, even though we've summarized the forms of secret codes by combining AOSP and manually extracted vendor-specific features, it remains challenging to cover all vendors' secret codes. If developers do not follow reasonable naming patterns when setting actions, our heuristic approach may miss the Intent related to secret codes. Moreover, in the extraction of the secret code, we did not analyze the native code, which could potentially also implement secret codes.

In addition, static analysis is limited in determining whether a secret code works on real devices, as apps often include conditions that rely on nested function calls and variable propagation. It leads to discrepancies between analysis results and real device behavior: some codes may appear executable under static analysis but remain untriggered on actual devices. While this constraint does not affect the detection of redundant secret codes, we still rely on manual testing to verify risky codes to ensure the accuracy of our results.

External Validity. To support secret code handling, manufacturers can define custom Broadcast Intent actions. In our analysis, we manually identified and summarized the custom actions used by 18 major vendors. However, for devices from other manufacturers, it remains necessary to manually inspect for custom Intent actions when analyzing secret codes. Otherwise, some vendor-specific codes may be missed.

Construct Validity. We use the LLM to classify secret codes and assess their redundancy in released mobile devices. We construct prompts using available information extracted from the secret code invocation flow. However, such input information is not always sufficient for the LLM for accurate LLM secret code classification and redundancy inference. In addition, even we use a voting method, the LLM may produce hallucinated content that cannot be fully eliminated.

VII. Related Work

OEM Customization Security. Existing research revealed the security issues introduced by Android customizations. Wu et al. [37] analyzed firmware and found that most cases of over-privileged apps and vulnerabilities stemmed from vendor modifications rather than the original AOSP code. Elsabagh et al. [24] proposed the FirmScope to identify numerous privilege escalation vulnerabilities in pre-installed apps which expose the large attack surface. Possemato et al. [30] extract the customization layers of each ROM and evaluate them using several metrics. Hou et al. [25] further supported these findings through large-scale firmware measurements, providing solid evidence of the security risks posed by customization.

Researchers also uncovered attack vectors introduced by vendor customizations. Zhou et al. [43] developed the ADDICTED tool, which revealed multiple vulnerable drivers caused by insufficient protection. Aafer et al. [19] performed differential analysis on customized ROMs and identified many exploitable inconsistencies resulting from altered security features. El-Rewini et al. [23] introduced the concept of "Residual APIs", which are often neglected in maintenance and are prone to access control vulnerabilities. At the same time, to automate the discovery of such deep-seated vulnerabilities, many static [32], [42], [44], [35], [36] and dynamic [26], [38], [39], [23], [21] analysis methods have been developed.

Analysis of Hidden Features. Prior research has revealed the prevalence and security implications of hidden behaviors in Android apps. Shan et al. [34] characterized self-hiding behaviors (SHBs). Similarly, Pham et al. [29] proposed HideMyApp, which shows that app fingerprinting via installed app lists is a widespread threat. Chen et al. [20] focused on hidden privacy settings, revealing that over one-third of apps often defaulting to permissive data sharing. Yang et al. [40] and Diao et al. [22] explored the misuse of non-SDK and accessibility APIs, both frequently exploited for stealthy operations or privileged access. Pourali et al. [31] presented ThirdEye to detect extensive use of multi-layer encryption and insecure custom protocols. Meanwhile, Sun et al. [35] and Samhi et al. [33] introduced HiSenDroid and Difuzer respectively. Their work demonstrated that attackers often obfuscate or delay the execution of privacy-violating code to evade analysis.

While the security risks of secret codes have been previously studied, prior research largely consisted of isolated vulnerability case reports. For instance, a pre-installed engineering app on OnePlus devices could grant full root access [1], and fuzzing tools revealed specific vulnerabilities like unauthorized factory resets [2]. While these studies highlight severe risks, our research provides the first comprehensive, multi-vendor analysis, uncovering more hidden secret codes and systematically characterizing this under-documented attack surface.

VIII. Conclusion

In this work, we conducted a large-scale study on the customization of secret codes in the Android ecosystem. We built a dataset of 673 firmware images from 18 different brands to analyze and explain how manufacturers implement and use secret codes. We summarized the usage patterns of secret codes in different functional categories, revealed that redundant codes are commonly found in multiple brands, and uncovered the trend in the development of secret codes. Then, we further validated our findings on seven real-world devices and identified three vulnerabilities that allow bypassing verification when enabling ADB debugging. Our research suggests the need for device manufacturers to enhance the review of secret codes, improving the stability and security of devices.

Data Availability

We have provided the prototype implementation on GitHub: https://github.com/Lynnrya/SecretCodes. For security reasons, the specific secret codes extracted are not disclosed.

Acknowledgements

We thank the insightful review comments. This work was supported by Natural Science Foundation of Shandong Province (Grant No. ZR2023MF043 and ZR2025MS991). Fenghao Xu was supported in part by the Start-up Research Fund of Southeast University (Grant No. RF1028624144).

References

[1] Heads Up: OnePlus Phones Have a Secret Root Backdoor and the Password Is 'angela'. https://www.theregister.com/2017/11/14/oneplus_backdoor/, 2017. (Accessed on 07/07/2025).

[2] The Secret Codes Tell the Secrets. https://conference.hitb.org/hitb-lockdown002/sessions/the-secret-codes-tell-the-secrets/, 2020. (Accessed on 07/07/2025).

[3] Androguard. https://github.com/androguard/androguard, 2025. (Accessed on 06/20/2025).

[4] Android Dumps. https://dumps.tadiphone.dev/dumps, 2025. (Accessed on 06/20/2025).

[5] AOSP. https://source.android.com/, 2025. (Accessed on 06/22/2025).

[6] Best Practices for Unique Identifiers. https://developer.android.com/identity/user-data-ids, 2025. (Accessed on 06/30/2025).

[7] BroadcastReceiver. https://developer.android.com/reference/android/content/BroadcastReceiver, 2025. (Accessed on 15/07/2025).

[8] DeepSeek. https://www.deepseek.com/, 2025. (Accessed on 14/07/2025).

[9] Google. https://developers.google.com/android/ota, 2025. (Accessed on 10/05/2025).

[10] Meizu. https://flyme.cn/firmware, 2025. (Accessed on 10/05/2025).

[11] OPPO. https://support.oppo.com/in/software-update, 2025. (Accessed on 10/05/2025).

[12] Redmi. https://roms.miuier.com/, 2025. (Accessed on 10/05/2025).

[13] Ro Build Tags. https://cs.android.com/android/platform/superproject/main/+/main:build/make/core/config.mk, 2025. (Accessed on 16/07/2025).

[14] Ro Build Type. https://source.android.com/docs/setup/build/building?hl=en#build-variants, 2025. (Accessed on 16/07/2025).

[15] Secret Code for All Mobiles. https://sites.google.com/view/secret-codes-for-all-mobiles/home, 2025. (Accessed on 06/30/2025).

[16] SpecialCharSequenceMgr.java. https://android.googlesource.com/platform/packages/apps/Dialer/+/91197049c458f07092b31501d2ed512180b13d58/src/com/android/dialer/SpecialCharSequenceMgr.java, 2025. (Accessed on 15/07/2025).

[17] TelephonyManager.java. https://android.googlesource.com/platform/frameworks/base/+/main/telephony/java/android/telephony/TelephonyManager.java, 2025. (Accessed on 15/07/2025).

[18] Unstructured Supplementary Service Data. https://sinch.com/glossary/ussd-meaning/, 2025. (Accessed on 06/30/2025).

[19] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *Proceedings of the 25th USENIX Security Symposium (USENIX-SEC), Austin, TX, USA, August 10–12, 2016*, 2016.

[20] Yi Chen, Mingming Zha, Nan Zhang, Dandan Xu, Qianqian Zhao, Xuan Feng, Kan Yuan, Fnu Suya, Yuan Tian, Kai Chen, XiaoFeng Wang, and Wei Zou. Demystifying Hidden Privacy Settings in Mobile Apps. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P), San Francisco, CA, USA, May 19-23, 2019*, 2019.

[21] Abdallah Dawoud and Sven Bugiel. Bringing balance to the force: Dynamic analysis of the android application framework. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS), virtually, February 21-25, 2021*, 2021.

[22] Wenrui Diao, Yue Zhang, Li Zhang, Zhou Li, Fenghao Xu, Xiaorui Pan, Xiangyu Liu, Jian Weng, Kehuan Zhang, and XiaoFeng Wang. Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Chaoyang District, Beijing, China, September 23-25, 2019*, 2019.

[23] Zeinab El-Rewini and Yousra Aafer. Dissecting Residual APIs in Custom Android ROMs. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, Republic of Korea, November 15–19, 2021*, 2021.

[24] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *Proceedings of the 29th USENIX Security Symposium (USENIX-SEC), August 12-14, 2020*, 2020.

[25] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. Large-scale Security Measurements on the Android Firmware Ecosystem. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, May 25-27, 2022*, 2022.

[26] Sudesh Kumar, Lakshmi Jayant Kittur, and Alwyn Roshan Pais. Attacks on Android-Based Smartphones and Impact of Vendor Customization on Android OS Security. In *Proceedings of the 16th International Conference on Information Systems Security (ICISS), Jammu, India, December 16-20, 2020*, 2020.

[27] Rui Li, Wenrui Diao, Zhou Li, Jianqi Du, and Shanqing Guo. Android custom permissions demystified: From privilege escalation to design shortcomings. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P), San Francisco, CA, USA, 24-27 May 2021*, 2021.

[28] Pei Liu, Yanjie Zhao, Mattia Fazzini, Haipeng Cai, John Grundy, and Li Li. Automatically detecting incompatible android apis. *ACM Transactions on Software Engineering and Methodology*, pages 15:1–15:33, 2024.

[29] Anh Pham, Italo Dacosta, Eleonora Losiouk, John Stephan, Kévin Huguenin, and Jean-Pierre Hubaux. HideMyApp: Hiding the Presence of Sensitive Apps on Android. In *Proceedings of the 28th USENIX Security Symposium (USENIX-SEC), Santa Clara, CA, USA, August 14-16, 2019*, 2019.

[30] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android OEM compliance and customization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P), San Francisco, CA, USA, 24-27 May 2021*, 2021.

[31] Sajjad Pourali, Nayanamana Samarasinghe, and Mohammad Mannan. Hidden in Plain Sight: Exploring Encrypted Channels in Android Apps. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, November 7-11, 2022*, 2022.

[32] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering, (ICSE), Pittsburgh, PA, USA, May 25-27, 2022*, 2022.

[33] Jordan Samhi, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Difuzer: Uncovering Suspicious Hidden Sensitive Operations in Android Apps. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, May 25-27, 2022*, 2022.

[34] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. Self-hiding Behavior in Android Apps: Detection and Characterization. In *Proceedings of the 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, May 27 - June 03, 2018*, 2018.

[35] Xiaoyu Sun, Xiao Chen, Li Li, Haipeng Cai, John Grundy, Jordan Samhi, Tegawendé F. Bissyandé, and Jacques Klein. Demystifying Hidden Sensitive Operations in Android Apps. *ACM Transactions on Software Engineering and Methodology*, 2023.

[36] Jikai Wang and Haoyu Wang. Nativesummary: Summarizing native binary code for inter-language static analysis of android apps. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, September 16-20, 2024*, 2024.

[37] Lei Wu, Michael C. Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013*, 2013.

[38] Xiaobo Xiang, Ren Zhang, Hanxiang Wen, Xiaorui Gong, and Baoxu Liu. Ghost in the binder: Binder transaction redirection attacks in android system services. In *Proceedings of the CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, Republic of Korea, November 15 - 19, 2021*, 2021.

[39] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. Atlas: Automating cross-language fuzzing on android closed-source libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, September 16-20, 2024*, 2024.

[40] Shishuai Yang, Rui Li, Jiongyi Chen, Wenrui Diao, and Shanqing Guo. Demystifying android non-sdk apis: Measurement and understanding. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, May 25-27, 2022*, pages 647–658, 2022.

[41] Dongsong Yu, Guangliang Yang, Guozhu Meng, Xiaorui Gong, Xiu Zhang, Xiaobo Xiang, Xiaoyu Wang, Yue Jiang, Kai Chen, Wei Zou, Wenke Lee, and Wenchang Shi. SEPAL: towards a large-scale analysis of seandroid policy customization. In *Proceedings of the 30th International World Wide Web Conference (WWW), Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, 2021.

[42] Yanjie Zhao, Li Li, Kui Liu, and John C. Grundy. Towards automatically repairing compatibility issues in published android apps. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, May 25-27, 2022*, 2022.

[43] Xiao-yong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (IEEE S&P), Berkeley, CA, USA, May 18-21, 2014*, 2014.

[44] Yuhao Zhou and Wei Song. Ddldroid: Efficiently detecting data loss issues in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, USA, July 17-21, 2023*, 2023.