Same App, Different Behaviors: Uncovering Device-specific Behaviors in Android Apps

Zikan Dong*
Beijing University of Posts and Telecommunications
Beijing, China

Yanjie Zhao*
Huazhong University of Science and Technology
Wuhan, China

Tianming Liu
Monash University
Melbourne, Australia

Chao Wang
Huazhong University of Science and Technology
Wuhan, China

Guosheng Xu
Beijing University of Posts and Telecommunications
Beijing, China

Guoai Xu†
Harbin Institute of Technology, Shenzhen
Shenzhen, China

Lin Zhang†
The National Computer Emergency Response Team/Coordination Center of China (CNCERT/CC)
Beijing, China

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

The Android ecosystem is significantly challenged by fragmentation, arising from diverse system versions, device specifications, and manufacturer customizations. The growing divergence among devices leads to marked variations in how a given app behaves across diverse devices. This is referred to as device-specific behaviors. Fragmentation not only complicates development processes but also impacts the overall industry by increasing maintenance costs and potentially harming user experience due to inconsistent app performance. In this work, we present the first large-scale empirical study of device-specific behaviors in real-world Android apps. We have designed a three-phase static analysis framework to accurately detect and understand the device-specific behaviors. Upon employing our tool on a dataset comprising more than 20,000 apps, we detected device-specific behaviors in 2,357 of them. By examining the distribution of device-specific behaviors, our analysis revealed that apps within the Chinese third-party app market exhibit more such behaviors compared to their counterparts in Google Play. Additionally, these behaviors are more likely to feature dominant brands that hold larger market shares. Reflecting this, we have classified these device-specific behaviors into 29 categories based on the functionalities implemented, providing a structured insight that is crucial for developers and stakeholders in the industry. Beyond the common behaviors, such as issue fixes and feature adaptations, we have observed 33 aggressive apps, including popular ones with millions of downloads. These apps abuse system properties of customized ROMs to obtain user-unresettable identifiers without requiring any permissions, posing significant privacy risks. Finally, we investigated the origins of device-specific behaviors, highlighting the significant challenges developers encounter in implementing them comprehensively. Our research aims to inform and equip industry practitioners with knowledge to enhance user experience and user privacy, marking a critical step toward addressing the less touched yet vital aspect of device-specific behaviors in the Android ecosystem.

ACM Reference Format:
Zikan Dong, Yanjie Zhao, Tianming Liu, Chao Wang, Guosheng Xu, Guoai Xu, Lin Zhang, and Haoyu Wang. 2024. Same App, Different Behaviors: Uncovering Device-specific Behaviors in Android Apps. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3691620.3695272

1 INTRODUCTION

Android stands out as a highly popular mobile operating system, commanding a market share of up to 71.4% [19]. The success of Android is closely linked to its open nature, yet this openness has given rise to notable fragmentation challenges emanating from the diverse array of devices and varying Android system versions. In contrast, the closed nature of the iOS ecosystem facilitates a cohesive user experience, with 17 major releases to date supported on just over 40 smartphone models [39, 52]. The scenario in the Android ecosystem is notably intricate. To elaborate, there exist over 24,000 distinct devices operating on the Android system [6], featuring diverse hardware configurations. Even when restricting the focus to those currently trending, the number of Android devices surpasses that of iOS devices by a considerable margin. Moreover, despite the Android system having released only 13 major versions

*Zikan Dong and Yanjie Zhao contributed equally to this research.
†Guoai Xu (xga@hit.edu.cn) and Lin Zhang (zhanglin@cert.org.cn) are the corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695272

thus far, certain prominent device manufacturers (such as Samsung, Huawei, and Xiaomi) extensively modify the Google-led Android Open Source Project (AOSP) to appeal to a broader consumer base. During the initial phases of system customization, manufacturers concentrated on enhancing the user interface, evident in the incorporation of "UI" in the names of numerous customized systems. Yet, as technology progressed, manufacturers began to progressively alter system functionalities and introduce new features absent in AOSP. These factors contribute to an increasing divergence among different Android devices. The challenge of fragmentation presents hurdles for the whole of industry. Developers need to invest more effort to develop, test, and maintain, ensuring a consistent user experience for their app across a diverse range of devices and system versions. Consequently, a single app may display distinct behaviors across various devices. Our research centers around this phenomenon, which we term ***device-specific behavior***.

Previous research [44, 46, 47] has shed light on device-specific issues in the Android ecosystem, primarily focusing on compatibility challenges. These efforts have led to significant advancements, including the development of automated tools like FicFinder [46] and Pivot [47], which identify device-specific compatibility issues by detecting patterns and extracting API-device correlations. However, beyond **compatibility issue fix**, which typically involve resolving hardware-specific or OS-specific glitches, there exists a critical category of device-specific behaviors—**manufacturer-introduced feature adaptation**. These behaviors leverage the innovative functionalities that manufacturers incorporate into their customized systems. While neglecting these adaptations may not lead to severe failures, embracing them can provide developers with the opportunity to attract a broader user base. Although existing works [46, 47] have touched upon this aspect, it has not been the main focus of their research.

The research gap becomes especially pertinent with **privacy-related** device-specific behaviors, a domain yet to be investigated in existing literature. When customizing their own Android systems, manufacturers may inadvertently undermine AOSP's security protection, opening new venues for malicious actors to gain access to sensitive user information unscrupulously. Reports [20] have indicated that these customizations have already been abused by certain popular apps. Specifically, by extensively collecting vulnerabilities in various major customized systems, the reported app, whose user base amounts to over 800 million, has achieved privilege escalation across numerous mainstream devices, subsequently gaining access to sensitive user data and silently monitoring users' every move. This situation poses significant challenges for industry stakeholders, including developers, security teams, and compliance officers, as they navigate an increasingly complex landscape of privacy risks and regulatory requirements.

Therefore, our research aims to provide a comprehensive understanding of device-specific behaviors in large-scale real-world Android apps, moving beyond the traditional lens of compatibility issues to encompass a broader spectrum of device-specific behaviors. We devised a static analysis pipeline to facilitate our large-scale study. Taking Android apps as input, we first perform static data flow analysis to identify code snippets that are associated with device-specific behaviors, and then provide a categorization of these device-specific behaviors based on the specific functionalities they implement.

Applying our pipeline to over 20,000 Android apps sourced from multiple app markets, we detected 2,357 apps exhibiting device-specific behaviors. In terms of distribution, we found that apps from Chinese app markets exhibit more device-specific behaviors compared to apps on Google Play, and device-specific behaviors are more likely to involve manufacturers with a larger market share. In terms of categorization, we have classified these device-specific behaviors into 29 types by their functionalities. These device-specific behaviors are predominantly implemented to fix compatibility issues or adapt to manufacturer-introduced new features. Additionally, we discovered privacy-related device-specific behaviors, revealing that 33 aggressive apps (including popular ones with millions of downloads) are abusing customized system properties to obtain user-unresettable identifiers without requiring any permissions, which should be protected by system-level permissions, posing a substantial threat to user privacy. We further investigated the sources of these device-specific behaviors and the channels through which developers can gather information related to device-specific behaviors. Notably, we encountered a scarcity of relevant information in the official documentation. Through our research, we raise developers' awareness of the broad spectrum of device-specific behaviors and provide guidance on code practices for implementing such behaviors. Additionally, we identified key deficiencies in the existing documentation on device-specific behaviors provided by manufacturers. Furthermore, our study acts as a call to action for manufacturers to implement stricter safeguards during the customization process, aiming to prevent the emergence of new security vulnerabilities.

In summary, this work offers these major contributions:

- We take the first step to present a comprehensive analysis of device-specific behaviors in large-scale real-world Android apps by implementing an automated static analysis framework, and we have successfully uncovered 2,357 apps with such behaviors.
- We make effort to characterize these device-specific behaviors, and create a taxonomy of 29 categories. Beyond the common practices including issue fix and feature adaptation, we have identified some aggressive behaviors that breach AOSP protection to obtain sensitive information unscrupulously, including some popular apps.
- Our research empowers developers with a more profound insight into device-specific behaviors, while underscoring a critical need within the industry for manufacturers to enhance relevant official documentation and fortify the security of system customization.

We make the source code of our tool, the summarized rules, and the dataset publicly available [11] to the research community.

2 BACKGROUND

2.1 Fragmentation Issues

Android fragmentation, arising from device variability and manufacturer customizations, poses significant challenges. Manufacturers leverage Android's open-source nature to customize system versions from AOSP, introducing new features like AI assistants [21]

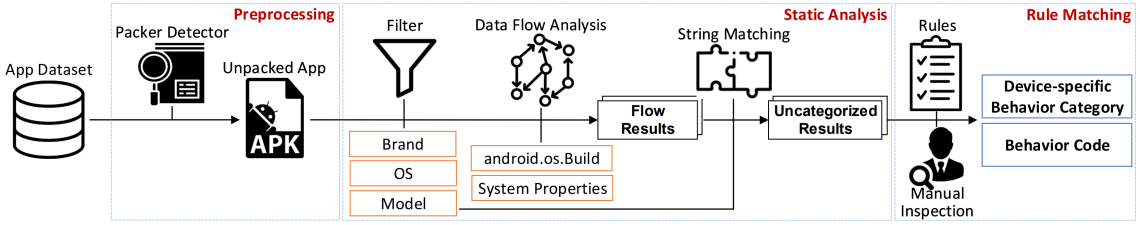


Figure 1: Study Methodology Design.

and mini window [30] to differentiate their devices. These alterations and customizations amplify distinctions among various devices, furthering the phenomenon of device-specific fragmentation.

While fragmentation offers choice to consumers, it complicates app development, as app behavior varies across devices, especially with customized UI (e.g., home screen and system bar) and hardware components (e.g., camera and Bluetooth). Ensuring seamless performance across diverse system and hardware configurations on Android devices, and harnessing the novel features in customized systems, demands developers to dedicate considerable time and financial resources to the development and testing process.

2.2 Privacy-related System Customizations

A unique class of system customizations pertains to privacy-related modifications. As certain manufacturers extensively customize the AOSP system, these customizations can extend to components associated with user privacy, including components linked to device identifiers or user location information. When issues emerge in such customizations, they introduce additional security risks to the customized system, consequently jeopardizing user privacy. Manufacturers might exhibit delays in adopting Google's measures aimed at safeguarding user privacy [38]. An illustrative instance is observed in Android 10, where Google restricts third-party apps from accessing non-resettable identifiers like the International Mobile Equipment Identity (IMEI). Nevertheless, LG's system did not promptly revise the access control policy for custom IMEI access APIs, resulting in a scenario where apps can still obtain the IMEI of LG devices even under the Android 10 system. Moreover, insufficient security measures may result in the exposure of user privacy data through newly introduced features by manufacturers [40]. In the case of the customized weather app by Vivo, the absence of security protection permits the app to access the user's accurate location without the necessity of seeking any permissions. Unscrupulous developers may exploit vulnerabilities in these customized systems through reverse engineering, potentially compromising user privacy data that ought to be safeguarded.

3 APPROACH

To gain a holistic understanding of device-specific behaviors in actual Android apps, we meticulously designed our research approach, as illustrated in Figure 1. The process takes an APK (i.e., Android Package) file as input and outputs the app's device-specific behavior category (e.g., user interface and permission management) along with the corresponding behavior code. Our approach contains three key steps: (1) *Preprocessing*, which detects whether apps employ packing techniques, and selects unpacked apps as analysis targets; (2) *Static Analysis*, which uses data flow analysis to identify code snippets related to device-specific behaviors; (3) *Rule Matching*, which involves categorizing the results from the second step using rules derived from manual inspection and generating an analysis result, including behavior category and relevant code.

3.1 Preprocessing

In order to enhance the security of apps, some developers may employ app packing techniques, which involve anti-static analysis (e.g., encryption, obfuscation and virtual machine-based protection) and anti-dynamic analysis (e.g., running environment detection, anti-debugging and seizing debugging interface) and anti-tampering (e.g., file integrity verification) [36, 37, 49].

App packing techniques can effectively prevent program analysis and cracking all existing packing solutions remains a formidable challenge. Therefore, our analysis focuses exclusively on unpacked apps and filters out packed ones. We use established packing detection tools [8] to verify the packing status of apps by examining featured files or package names within the APK files.

3.2 Static Analysis

We utilize static analysis techniques to analyze device-specific behaviors in apps. This approach enables a comprehensive examination of device-specific behaviors within apps without the necessity of preparing numerous devices from diverse brands and models.

Observations based on pilot study. Our objective is to detect all methods that may encompass device-specific behaviors. To accomplish this, we initially outline the pattern of device-specific behaviors. After conducting a reverse analysis of more than 100 popular real-world apps, with a focus on calls to potential system methods associated with device-specific behaviors, we have derived the following observations:

Table 1: The Method to Obtain Device Information.

Field in android.os.Build	System Property	Description
BRAND	ro.product.brand	Consumer-visible Brand
DEVICE	ro.product.device	Name of the Industrial Design
DISPLAY	ro.build.display.id	Build ID String for Users
FINGERPRINT	ro.build.fingerprint	String that Identifies Current Build
MANUFACTURER	ro.product.manufacturer	Manufacturer of the Product/Hardware
MODEL	ro.product.model	End-user-visible Name for the Product
PRODUCT	ro.product.name	Name of the Overall Product

(a) Prior to executing device-specific behaviors, the app must retrieve current device information, such as the brand and specific device model. Based on our review of the existing literature [44, 46, 47] and related methods within the Android system, we identify two primary methods for retrieving device information. One method involves accessing fields within the `android.os.Build` class, which

provides information sourced from system properties about the current environment. The alternative method involves directly retrieving the values of system properties associated with the device information, typically achieved by invoking methods in the `android.os.SystemProperties` class through Java reflection. We summarize the relevant methods in Table 1. (b) After acquiring the current device information, the app assesses whether the current environment necessitates the execution of device-specific behaviors. Consequently, the code related to device-specific behaviors contains identifiers (typically in string form) for devices requiring special handling by the app, including specific brands, operating systems, and models. These identifiers are subsequently employed in comparisons with the current device information. (c) When comparing the current device information with target device identifiers, apps often employ"if statement" and string processing functions, such as `equal`, `toLowerCase/toUpperCase`, and `startsWith/endsWith`. To conclude, the device-specific behavior code should include "if statements", with the original or processed device information incorporated into the conditions of these statements.

Methodological steps in detail. Based on our observations, we formulate the following static analysis process. First, we constructed a database containing various globally available device information. We crawled GSMArena [13], a popular online platform focused on mobile technology and smartphones, which includes detailed information about devices from different manufacturers. Specifically, the information we collected includes 72 brand names (e.g., Samsung, Xiaomi, and Huawei), 33 operating system names (e.g., One UI, MIUI, and HarmonyOS), and 8,032 mobile model names (e.g., SM-S918B represents Samsung Galaxy S23 Ultra).

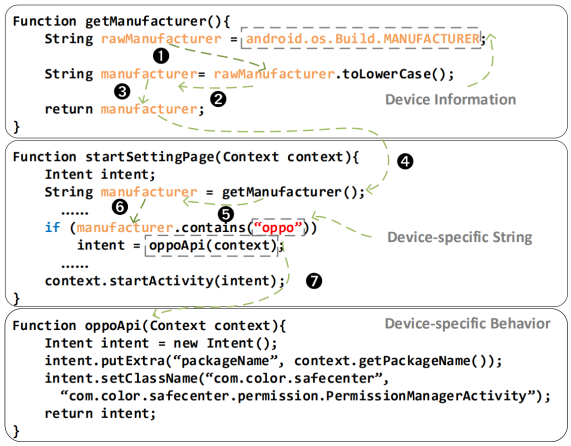


Figure 2: Data Flow Analysis.

Then, we aim to conduct inter-procedural data flow analysis. We decompile APK files and transform Java code into `Jimple`, an intermediate representation used in Soot, and generate Inter-process Control Flow Graph (ICFG) for apps to obtain the function call relationships. Our approach is built on Soot [45] and FlowDroid [34], both of which are widely used Android static analysis tools. We manage to identify all methods capable of accessing current device information according to our first observation. To be more specific, we start with traversing all methods in the app to locate the two methods (i.e., fields in `android.os.Build` class and system properties). After identifying a method that acquires device information, we conduct an intra-procedure def-use analysis on variables storing device information to find all uses of the variable within the method, as shown in Steps ❶ and ❸ in Figure 2. The example in Figure 2 illustrates the process of starting the permission settings page. Firstly, the `startSettingPage` function obtains the current device manufacturer through the `getManufacturer` function. After determining the current device brand is OPPO, it invokes the `oppoApi` function to navigate to OPPO's permission settings page.

Further, we proceed with inter-procedural analysis when encountering the following scenarios: (a) When device information is passed as a parameter to a function call, we employ different strategies for inter-procedural data flow analysis depending on the type of called function. In instances where the called methods have concrete bodies (i.e., containing specific code rather than being abstract methods from libraries), we perform intra-procedural data flow analysis on the corresponding parameters in the method. In cases where the called methods lack concrete method bodies, such as Java library methods (e.g., string processing methods), we skip the analysis of these methods. Instead, we directly perform intra-procedural data flow analysis on the method's return value within the original method, as illustrated by Step ❷ in Figure 2. (b) When device information or its variants are used as a method's return value, such as Step ❸ in our example. We use the ICFG to identify all methods calling the method that return device information, as depicted in Step ❹. Subsequently, we perform intra-procedural data flow analysis on the variables corresponding to the method's return values within these calling methods, as shown in Steps ❺ and ❻.

Having obtained all the methods capable of accessing device information, we aim to identify if statements used to compare the current device information with the target device identifiers based on our second and third observations. With the results of the aforementioned data flow analysis, we can determine whether the current device information appears within the condition of an if statement, as illustrated in Step ❻ in Figure 2. Next, we check whether the target device identifier is present around found if statements using string matching and our device information database.

After confirming that a specific if statement compares the current device information with the target device identifier, we conclude that the branches controlled by the if statement contain device-specific behaviors. If there are methods invoked within the branch, and the called methods have concrete bodies, we further traverse the called methods and consider them as device-specific behaviors, as shown in Step ❼ in Figure 2. When new method calls still can be found during this process, we continue the deepening process until no new method calls are discovered. Consequently, we identify all code snippets related to device-specific behaviors.

3.3 Rule Matching

After isolating the code snippets related to device-specific behaviors, we adopt a rule-matching methodology for categorizing these behaviors based on the functionality implemented by the code. After the static analysis phase, we encountered an extensive amount of code snippets associated with device-specific behaviors, making it challenging for individual inspection of each snippet. Nevertheless,

Table 2: Examples of the 29 Categories of the Identified Rules.

Behavior Type	Category	Keyword	Behavior Type	Category	Keyword	Behavior Type	Category	Keyword
Compatibility Issue Fix	Memory Leak	android.view.inputmethod.InputMethodManager	Manufacturer-introduced Feature Adaptation	Permission Management	com.vivo.permissionmanager	Privacy-related	OAID	com.huawei.hwid
		android.gestureboost.GestureBoostManager			com.miui.securitycenter			com.asus.msa
	Malfunctional Features	android.webkit.WebSettings		Foldables	hardware.sensor.posture		SystemProperties Containing Hardware Identifiers	ro.meizu.hardware.imei
		android.bluetooth.BluetoothAdapter		DisplayCutouts	flyme.config.FlymeFeature			ro.ril.miui.meid

we observe that certain device-specific behaviors, such as addressing prevalent issues or adapting to common features, frequently recur across apps. In light of this, we have implemented a clustering method to facilitate our categorization.

We cluster code snippets that invoke the same system methods based on the understanding that (a) apps must use system methods to implement system functionalities, (b) code snippets calling the same system methods share similar functionalities, and (c) these method calls typically remain unchanged after code obfuscation. Specifically, for each code snippet, we extract system method calls based on package names prefix (e.g., "android.", "androidx."), then concatenate these calls into a single string to serve as a feature descriptor. We exclude snippets with fewer than four system calls to prevent inaccurate groupings. Snippets with the same characteristics are grouped into clusters. After the clustering, we sample code snippets from each cluster and manually select keywords that best represent the functionality of the cluster to formulate our rule. These keywords are generally sourced from method names, strings containing method names (commonly used in intent construction or Java reflection), or other meaningful strings (e.g., customized system property names), considering the rich semantic information they provide. For example, in the case shown in Figure 2, the string `com.color.safecenter.permission.PermissionManagerActivity` is chosen as the keyword, which is the activity name corresponding to the OPPO permission settings page. Each selected keyword is then manually labeled with a functionality category (e.g., permission) to formulate the rules in the form of a key-value pair, i.e., `[Category:Keyword]`. If the code snippet contains the relevant keyword, we categorize it into the corresponding category.

The rule-labeling process was primarily led by two experienced Android researchers, with a third resolving disputes. Specifically, the two lead researchers independently reviewed the same code snippets and formulated rules. Prior discussions on the methods for rule formulation—such as prioritizing method names as keywords—were conducted to reduce disagreements. They then convened to discuss and reconcile their findings. Rules that were mutually agreed upon were adopted. In cases of discord, the researchers would consult the third researcher until a consensus was reached.

In total, after manually analyzing the results of all Android apps in our dataset, we labeled 170 keyword rules corresponding to 29 distinct categories. Details of our dataset are provided in §4 . It is common for one functionality category to correspond to multiple keywords. We list some of the rule examples in Table 2, with the detailed breakdown available on our website [11]. These categories are aligned with the three major behavior types outlined in the Introduction, i.e., compatibility issue fix (e.g., memory leak and malfunctional features), manufacturer-introduced feature adaptation (e.g., permission management and UI-related adaptations), and privacy-related device-specific behaviors (e.g., system properties leaking hardware identifiers). We elaborate them in §4.4.

4 RESULTS

4.1 Dataset

To explore device-specific behaviors in large-scale real-world Android apps, we amassed 21,889 recently updated apps from Google Play and two third-party app markets, namely App China [9] and Huawei AppGallery [14], each with a unique hash value. In our selection process, we prioritized apps based on their update recency, under the assumption that apps updated more recently are more likely to exhibit device-specific behaviors. Given Google Play's dominance in the global market, we first sourced Android apps directly from it, utilizing the widely recognized AndroZoo dataset [33]. We focused on the most recently updated apps from this collection. Another driving factor was the unavailability of Google services in China, coupled with the significant presence of prominent device manufacturers from China. This prompted us to conduct a comprehensive global study on device-specific behaviors, and include App China in our analysis. App China, a dedicated Android app market, is focused on serving users in China with a substantial number of apps available in AndroZoo [17]. Furthermore, to explore potential brand biases in manufacturer-organized app markets, where apps may receive more attention from manufacturer systems, we selected the Huawei AppGallery as a representative case for investigation. We employed crawlers to gather apps from two third-party markets, and the details of app distribution and device-specific behaviors are shown in Table 3.

4.2 Research Questions

In our exploratory investigation aimed at comprehending the current state of device-specific behaviors in real-world Android apps, we strive to address the following three research questions (RQs):

- RQ1 What is the distribution of device-specific behaviors in real-world Android apps? Does their prevalence vary significantly across different app markets? What is the distribution of the functionalities associated with these behaviors?
- RQ2 What functionalities do these device-specific behaviors implement?
- RQ3 Where do these device-specific behaviors come from? From the developers' standpoint, how can they acquire knowledge about these device-specific behaviors?

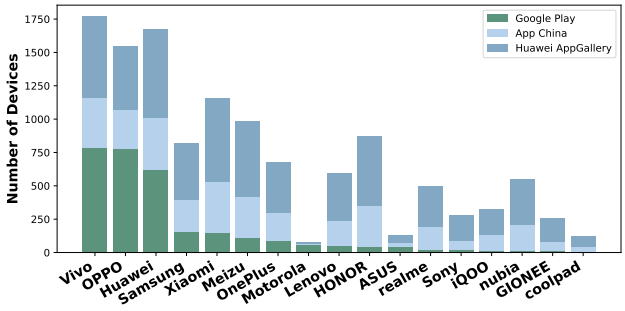
4.3 RQ1: Distribution of Device-specific Behaviors

To investigate the distribution of device-specific behaviors, we applied our research approach to 16,139 unpacked real-world Android apps in our dataset. We successfully analyzed 7,079, identifying device-specific behaviors in 2,357 apps. Our experiment was unable to analyze over half of the collected apps, primarily due to the complexity of real-world apps from app markets and the limitations of the static analysis tools used in our experiment, as mentioned

Table 3: Distribution of Apps Collected and Unpacked Apps across Markets.

	# Apps Collected	# Unpacked Apps	# Apps with Behaviors
Google Play	12,469	12,255	1,147
App China	3,120	1,758	443
Huawei AppGallery	6,300	2,126	767
Total	21,889	16,139	2,357

in previous studies [35, 50]. In [50], when analyzing apps from Google Play using FlowDroid, the analysis was successful for 56% of the apps, yielding results similar to ours. The static analysis tool encounters specific step analysis failures, and upon such exceptions, it retries the steps, resulting in an infinite loop that reaches the 1-hour timeout we set. Regarding the proportion of apps exhibiting device-specific behaviors, the two app markets from China (i.e., 70% in Huawei AppGallery and 77% in App China) are significantly higher than that of Google Play (i.e., 21% in Google Play). This suggests that developers from China tend to incorporate more device-specific behaviors into their apps.



Note: from left to right in the figure, the device-specific behaviors are sorted by the frequency of occurrence in Google Play apps. This facilitates a comparison of the differences between Google Play and the other two app markets. The sorting method used in Figure 4 and Figure 5 is the same as in this figure.

Figure 3: Distribution of Brands Involved in Device-specific Behaviors.

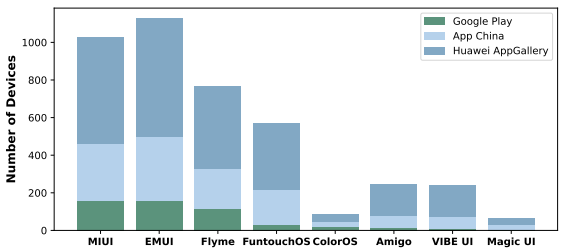


Figure 4: Distribution of OSes Involved in Device-specific Behaviors.

Furthermore, in Figure 3 and Figure 4, we summarized the distribution of brands and operating systems involved in these device-specific behaviors. In general, brands with high market share and their corresponding operating systems are more likely to appear in device-specific behaviors. Apps from two Chinese markets tend to implement device-specific behaviors for a broader range of brands and customized systems. The notable observation is that the prevalence of Vivo, OPPO, and Huawei devices in Google Play apps surpasses that of other brands significantly. This is because many Google Play apps tested leverage the customized audio features in the systems of these three manufacturers when developing audio-related functionalities. Further, we did not observe such behavior in apps from the two Chinese markets, as illustrated in the first column of Figure 5. Moreover, through a comparison of the results from App China and the Huawei AppGallery, we did not observe significant brand bias. It's worth noting that we have excluded brands and operating systems that appeared less than 20 times in Figure 3 and Figure 4. We did not observe many device-specific behaviors related to Huawei's HarmonyOS because identifying the HarmonyOS system requires dynamically invoking custom APIs within the HarmonyOS system through Java reflection, which is beyond the capability of our approach.

In Figure 5, we have statistically analyzed the distribution of device-specific behaviors based on their functionalities. After a manual inspection in §3.3, we formulated a total of 29 rules, with each rule corresponding to a specific behavioral feature. The most prevalent device-specific behavior in Google Play apps is related to the previously mentioned audio features, which, however, appear infrequently in the other two third-party app markets. In terms of other functionalities, the differences between Google Play and other app markets are relatively smaller. Apps from the two third-party app markets exhibit more instances of device-specific behaviors. Additionally, we also calculated the average number of brands, systems, and models involved in apps with device-specific behaviors in each app store, as shown in Table 4. In the Chinese third-party app markets, a single app tends to include more device-specific behaviors, involving a greater variety of brands, systems, and models, and implementing a wider range of functionalities. This explains why, despite having fewer apps with device-specific behaviors in the two third-party markets compared to Google Play, they exhibit a higher diversity of device-specific behaviors.

Table 4: Average of Device Information and Functionalities in Apps with Device-specific Behaviors across App Market.

	Avg Brands	Avg OSes	Avg Models	Avg Functionalities
Google Play	2.904	0.493	1.150	2.267
App China	7.365	4.130	2.541	9.939
Huawei AppGallery	7.522	4.577	3.027	10.778

Answer to RQ1: *Device-specific behaviors are prevalent in real-world apps, especially in the two third-party app markets, where their prevalence surpasses that of Google Play. Mainstream brands exhibit these behaviors more frequently, and manufacturer-organized app markets show no brand bias.*

4.4 RQ2: Functionality Implemented through Device-specific Behaviors

To explore the functionalities of device-specific behaviors, we researched relevant information and analyzed each identified behavior type. Device-specific behavior can be broadly categorized into three major types, including 29 kinds of sub-categories.

4.4.1 Compatibility Issue Fix. Due to manufacturers' customization of device hardware and system components, certain devices

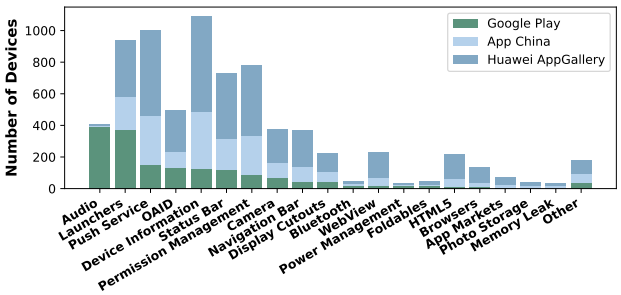


Figure 5: Distribution of Functionalities Implemented by Device-specific Behaviors.

may encounter specific issues. These issues are usually discovered in a particular system version and addressed in one or several subsequent system updates. While these issues are typically resolved within a relatively short period, the method of system updates does not guarantee that the solutions will reach every user's device in a timely manner. Not all users undertake system updates promptly, leading to a significant extension of the lifecycle of these issues. This necessitates app developers to address these device-specific issues. Device-specific issues can be further classified into two types: memory leak issues and malfunctional features.

Memory Leak. We have discovered that some device-specific behaviors aim to address memory leak issues in specific components of Huawei (i.e., EMUI) and Samsung systems in certain versions. Memory leak is a type of resource leak that occurs when a program does not timely release unneeded memory, leading to a gradual reduction in available system memory. In Android 7 EMUI 5, there is a memory leak issue in the EMUI system's customized component `android.gesturebоost.GestureBoostManager`. This component holds a reference to the app's activity and fails to release it when the app stops and garbage collection is needed, leading to memory leaks. Similarly, Samsung's system components `com.samsung.android.content.clipboard.SemClipboardManager` and `com.samsung.android.emergencymode.SemEmergencyManager` in Android 7, along with Huawei's `android.view.inputmethod.InputMethodManager` in Android 8, all contain memory leak issues. Developers must manually release references between problematic system components and app components (e.g., Activities) when they are destroyed, enabling proper garbage collection.

Malfunctional Features. Another type of device-specific issue is that certain system features fail to function properly on some devices. The system features refer to those present in the AOSP system, so these issues will significantly impact developers' normal development. We further categorize them based on features into the following categories:

(1) Camera. The camera, as a crucial component of smartphones, receives significant attention from manufacturers, who heavily customize the hardware. However, this has also led to many malfunctioning features. For example, Lenovo devices with the model X804F cannot reliably retrieve the front camera ID through the `android.hardware.Camera` class. The Nexus 5X and Mi 5 have an opposite rotation direction for photo previews compared to other devices. When setting the focus mode for the camera, some devices (i.e., MEIZU MX3 and Samsung Galaxy S4) cannot retrieve their supported focus modes through system APIs and must rely on developers to set the focus modes individually.

(2) Photo Storage. In the AOSP system and most devices, camera photos are stored in path "/sdcard/DCIM/Camera", while screenshots are stored in "/sdcard/Pictures/Screenshots". But Xiaomi has changed the storage path of screenshots to "/sdcard/DCIM/Screenshots", requiring developers to make separate modifications for Xiaomi devices when obtaining screenshot files through file paths.

(3) HTML5. The issues related to HTML5 mainly involve the HTML5 geolocation feature, which enables apps to obtain the device's location via JavaScript. On certain devices running specific versions of Android, this feature is unavailable, and it can't be detected through system APIs. Therefore, some developers have compiled a list of devices, along with corresponding system versions, (e.g., Vivo Xplay5A running Funtouch OS_2.5.1) where this feature is non-functional.

In other aspects, the overview mode of **WebView** on Meizu devices is different from that on other devices and does not function properly. In terms of **Bluetooth** functionality, some devices require special parameters during Bluetooth device scanning, and there are issues with the offloaded scan batching and offloaded filter functionality on certain devices. The **VPN** functionality on Samsung devices may ignore certain DNS servers, requiring apps to manually add routes for DNS servers.

4.4.2 Manufacturer-introduced Feature Adaptation. Device manufacturers build on Google's AOSP system by adding new features to their customized systems to boost product competitiveness. To fully utilize these manufacturer-introduced features, app developers must invest significant effort in adaptation, enhancing functionality, and improving the user experience.

Push Service. Push services enable apps to deliver message notifications to user devices, even when the device is locked, and the app is not actively running. Google offers the official push notification service, Firebase Cloud Messaging (FCM) [26], which is a widely adopted solution. However, in certain countries like China, where devices cannot connect to Google services, utilizing FCM for push notifications becomes unfeasible. Developers have the option to implement message push through long connections, where a lasting link is upheld between the device and the server. Furthermore, device manufacturers introduce their proprietary push services, such as Huawei Push Kit [23] and Mi Push [32]. Developers can integrate the manufacturer's push SDK into their app, enabling the creation of push notifications on the manufacturer's operating platform, which can also provide enhanced stability. Hence, we observed that developers incorporate push SDKs from prominent device manufacturers into their apps, employing the respective manufacturer's push service depending on the current system.

Power Management and Permission Management. In today's smartphone realm, battery life is paramount in assessing device performance. Manufacturers are increasing battery capacity and optimizing systems to reduce unnecessary code execution. Yet, strict power management policies pose challenges for developers. In stock Android, apps can persist in the background after being closed via the recent screen [24], initiating actions like receiving push notifications. Background activity only stops with force stop or manual battery optimization settings. Apps using Firebase for

push notifications won't receive them in these conditions. Conversely, customized systems force-stop apps upon recent screen closure, disallowing background operation by default. As a result, developers need to request auto-start permissions to ensure features like push messaging, a requirement not present in stock Android. Managing these permissions lacks an official method, requiring apps to navigate specific activities to guide users, as depicted in Table 5. Similarly, some manufacturers revamp permission settings pages (i.e., different activity) to handle custom permissions.

Table 5: The Activities for Managing App Auto-start Permissions.

Manufacturer	Activity for Managing Auto-start Permissions
Samsung	com.samsung.android.sm.ui.battery.BatteryActivity
Xiaomi	com.miui.permcenter.autostart.AutoStartManagementActivity
Huawei	com.huawei.systemmanager.startupmgr.ui.StartupNormalAppListActivity
OPPO	com.coloros.safecenter.permission.startup.StartupAppListActivity
Vivo	com.vivo.permissionmanager.activity.BgStartUpManagerActivity
Nokia	com.evenwell.powersaving.g3.exception.PowerSaverExceptionActivity
OnePlus	com.oneplus.security.chainlaunch.view.ChainLaunchAppListActivity

Launchers. Badge refers to a visual indicator displayed on the app icon on the home screen (i.e., launchers). It is typically a small number or a dot, providing a glance at the status of the app, such as the number of unread messages or notifications. However, the stock Android system does not support displaying numbers on the badge. Influenced by iOS, which supports displaying numbers in the badge, major device manufacturers have implemented the functionality to show numbers in the badge within their customized launchers. Developers need to call the corresponding customized APIs to modify the number in the badge.

Another issue related to the launchers is shortcuts [10] management. Different customized systems handle shortcuts in varying ways. Google and some Samsung devices allow apps to create shortcuts by default, while other manufacturers, like Huawei and Meizu, do not grant this permission. Apps need to adopt different methods to check shortcut installation permissions based on the system.

Display Cutouts and Foldables. A display cutout or notch refers to an area on some devices that extends into the display surface. Although manufacturers like Huawei, Xiaomi, and OPPO had already introduced devices with cutouts upon the release of Android 8, official support for them via the `DisplayCutout` class was included in Android 9. Initially, these manufacturers incorporated cutout adaptation into their customized Android 8 systems, providing position and dimensions for the cutouts. Subsequently, with the launch of Android 9, they shifted to the official adaptation method. However, since many users delay system updates, developers must implement diverse strategies based on the Android version to ensure proper cutout support. They rely on manufacturers' methods for Android 8 and adopt the official approach for versions post-Android 9. Another emerging design trend is foldable [16]. For effective utilization, apps need to adjust layouts based on foldable states (e.g., flat and half-opened). Manufacturers store this information in various ways (e.g., custom APIs and system properties).

Device Information. When manufacturers customize based on the AOSP system, similar to AOSP, they release multiple sub-versions within a major version. Manufacturers may address some issues in minor version updates. To access information about these issue fixes or simply to collect device information for analytics, developers need to obtain the version of the customized system. The version information is typically stored in customized system properties.

Beyond the aforementioned considerations, a parallel situation exists with the **Status Bar**, mirroring the challenges posed by display cutouts. Manufacturers introduced new styles for the status bar prior to Android, necessitating developers to employ diverse methods for distinct system versions. Conversely, certain manufacturers introduce novel **Navigation Bar** styles, including Meizu's smart bar. Apps must utilize customized functions to retrieve the status of the system bar and adapt the interface display accordingly. The manufacturers also add some new pre-installed apps, such as **Browsers**, **Video Players**, and **App Markets**. When needed, apps can leverage these more powerful customized apps. As mentioned in §4.3, some manufacturers provide additional **Audio** features, offering apps ear return functions.

4.4.3 Privacy-related. Further, some device-specific behaviors used to gather sensitive user data can raise potential privacy concerns.

OAID. Starting with Android 10, third-party apps are restricted from accessing non-resettable hardware identifiers like IMEI and serial numbers. As an alternative, Google recommends apps use advertising ID [2] for advertising and analytics purposes. However, in regions where access to Google services is restricted, Google's advertising ID also becomes unfeasible. To address this gap, in 2019, the China Mobile Security Alliance introduced the Open Anonymous Device Identifier (OAID) [1] as an alternative. Unlike Google's advertising ID, OAID's implementation varies by manufacturer and is embedded in their customized systems. Consequently, methods for obtaining OAID differ across devices, necessitating developers to adapt to these discrepancies. As each manufacturer independently implements OAID, there's a risk of vulnerabilities or inadequate security measures, potentially compromising user privacy [28].

```
public String getMeizuIMEI(Conetxt context){
  String IMEI;
  if (haveReadPhoneStatePermission(context)){
    return context.getSystemService("phone").getDeviceId();
  } else {
    Class<?> cls = Class.forName("android.os.SystemProperties");
    Method method = cls.getMethod("get", String.class);
    String propertyValue = (String) method.invoke(null, "ro.meizu.
        hardware.imei1");
    return propertyValue;
  }
}
```

Figure 6: Retrieving the IMEI of Meizu Devices through Customized System Properties.

System Properties Containing Hardware Identifiers. Manufacturers may accidentally introduce additional system properties that store sensitive information, potentially compromising user privacy. These system properties, accessible without requiring any permissions by default, become potential targets for malicious exploitation. We observed certain apps attempting to access the sensitive data stored in system properties introduced by the customized system, such as `ro.meizu.hardware.imei1`, `ro.vendor.vivo.serialno`, and `ro.ril.miui.meid`. Apparently, from their names, it appears that these properties contain user-unresettable hardware identifiers, which, in higher versions of Android, are inaccessible to third-party apps. As demonstrated in the code snippet in Figure 6, this app

manages to obtain the device's IMEI through the system property when the app is not granted permission to access IMEI (i.e., `android.permission.READ_PHONE_STATE`), infringing on user privacy. We identified a total of 33 apps exhibiting similar behaviors. Some apps are quite popular, e.g., "com.global.wt" gains 500K downloads on Google Play, "com.wibo.bigbang.ocr" have 2M downloads on Huawei AppGallery, and "com.fengbo.live" receives a significant 16 million downloads on Huawei AppGallery.

Furthermore, we attempted to verify these findings on real devices by utilizing cloud testing platforms [12, 31], as we lacked access to specific devices for direct testing. We developed an Android app that request no permissions and has the single functionality of retrieving all system properties from the device by executing the `getprop` command. Notably, we identified the existence of system property `ro.meizu.hardware.imei1` on multiple Meizu devices (e.g., Meizu Note9 running Android 9, Meizu 16s Pro running Android 9 and Meizu 18s Pro running Android 11) and confirmed the system property can be accessed without any permissions. In Android 9, such an exploit allows obtaining the device's IMEI without requiring permission or user consent. Even worse, despite Google's strict prohibition of third-party apps accessing the IMEI in Android 11, the exploitation of this customized system property enables third-party apps to freely retrieve the IMEI on devices from major manufacturers like Meizu, effectively circumventing the restrictions imposed by the Android platform. Additionally, we found that certain Meizu devices expose hardware identifiers like the Mobile Equipment Identifier (MEID) and serial number in system properties without requiring permissions. This circumvents Android's permission restrictions on apps, clearly infringing on user privacy.

Unfortunately, we were unable to confirm the exploitability of other system properties. In the cloud testing platform, we did not find any Vivo tablet devices, so we were unable to validate system property `ro.vendor.vivo.serialno`, which exists only in Vivo's tablet devices based on the identified device-specific behaviors. On Xiaomi devices, we did find the existence of system properties storing hardware identifiers (i.e., `ro.ril.miui.imei` and `ro.ril.miui.meid`). However, due to access controls set by Xiaomi for these system properties, we cannot retrieve the values of these properties through our Android app. It is worth noting that, since these system properties are stored in files, access control can be applied to them. Only with higher-level permissions like ADB [4], we can access the content of these properties.

Answer to RQ2: *Device-specific behaviors fall into three types: common compatibility issue fixes, manufacturer-introduced feature adaptations, and privacy-related behaviors. Surprisingly, a considerable number of aggressive apps in the market abuse the customized system properties to get hardware identifiers without requiring any permissions, even some of them have been downloaded for millions of times. We advocate community to pay special attention to such kinds of behaviors.*

4.5 RQ3: Sources of Device-specific Behaviors

To investigate the origin of these device-specific behaviors, we first extracted the package names of methods associated with these behaviors and recorded their frequencies. This analysis enabled us to distinguish whether these behaviors originated from SDKs or were embedded in the app's code by developers. It is worth noting that we manually excluded obfuscated package names whose sources cannot be identified. After this process, we obtained 879 distinct package names from 32,323 methods, and categorized the sources of behaviors into three types: general-purpose SDKs (`Type-I` SDKs), SDKs tailored to device-specific behaviors (`Type-II` SDKs), and developer-written code, as delineated in Table 6.

Table 6: Distribution of Behavior Sources and Top 3 Prominent Behaviors Types for Each Source.

	Type-I SDK	Type-II SDK	Developer Code
Involved Apps	1148 (49%)	1100 (47%)	411 (17%)
Top 3 Focused Behavior Type	Device Information (75%)	Audio (36%)	Status Bar (33%)
	Push (70%)	Launchers (29%)	Push (32%)
	Permission Management (50%)	Status Bar (22%)	Permission Management (22%)

`Type-I` SDKs are designed for general purposes, such as push notifications or content sharing (e.g., SDKs from major companies like Facebook or push SDKs like JPush [15]). They incorporate device-specific behaviors to ensure consistent functionality across various devices. As shown in Table 6, these SDKs primarily focus on device information collection across different customized systems, push services, and permission management. `Type-II` SDKs are specifically designed to assist developers in implementing a specific functionality more reliably across devices. Instances of Type-II SDKs encompass Agora SDK [3], designed for audio and video functionalities, ShortcutBadger SDK [7] for setting the number of unread message on home screen, and AutoStarter SDK [29] for requesting self-start permission for the app. These two types of SDKs constitute a majority of behavior sources, while the proportion of developer-implemented device-specific behaviors is relatively small (17%). We further discovered that in general, developers only manage to implement one or two types (an average of 1.8 types) of device-specific behaviors in their apps, which reflects the difficulty for developers to achieve such integration. Indeed, it is impractical for most developers to implement exhaustive device-specific behaviors across a wide range of device models and system versions. This spurred our investigation into the availability and accessibility of related information from the developers' perspective to assess the challenge.

Our case study focuses on the official documentation provided by three major manufacturers, namely Samsung [25], Xiaomi [18], and OPPO [22]. We browsed the official documentation and utilized its search functionality, attempting to find information about the 29 types of device-specific behaviors we identified. Given the limited content in the global versions of Xiaomi and OPPO documentation, we referred to their Chinese documentation. Among the **22** behaviors associated with Samsung devices, only information on **5** of them was found in Samsung's documentation. Xiaomi's documentation covered **9/18**, while OPPO's official website covered **10/17**. The available documentation predominantly highlights new features introduced by manufacturers, such as push services and foldables. Notably absent were details about compatibility issue fixes and comprehensive explanations of system customizations, such as modifications to screenshot storage locations or the addition of system properties. These documents primarily serve to guide developers in understanding the new features within customized systems and how to effectively leverage them. We further

extended our search to GitHub and search engines. Relevant information corresponding to our discoveries was located on technical forums like XDA Forums [5] and Stack Overflow [27], as well as on GitHub's issue descriptions and shared code snippets. The only exception was the absence of information about custom system properties containing hardware identifiers. In the retrieval process, we found that information related to device-specific behaviors is highly scattered. The keywords in §3.3 were instrumental in assisting our search, while it may be challenging for developers to locate relevant information without any prior knowledge. We will further discuss how our work can assist developers in this process in §5.

Answer to RQ3: *Device-specific behaviors in apps are primarily driven by general-purpose SDKs and those expressly designed to facilitate the implementation of such behaviors. Cases of developers coding these behaviors independently are relatively rare. Additionally, manufacturers' official documentation provides limited information on device-specific behaviors, which is often scattered online, posing challenges for developers to access relevant details.*

5 DISCUSSION

Implication. Our investigation reveals the widespread occurrence of device-specific behaviors within apps, which are essential for fixing compatibility issues, ensuring consistent performance across devices, and leveraging novel features introduced by manufacturers to augment app functionality.

However, many apps lack these behaviors, potentially leading to operational inconsistencies or missing features. This gap may stem from developers' limited awareness of device-specific issues and features, or uncertainties about how to address them. Our research provides the industry with crucial insights into these behaviors, identifying potential issues across devices and offering guidance on developing effective device-specific behaviors.

We observed that manufacturers' documentation often highlights new features but falls short in detailing device-specific issues and modifications to existing AOSP functionalities. We recommend that manufacturers adopt a more proactive approach by offering comprehensive information on device-specific issues, solutions, and system customizations, like permission management.

Moreover, our findings indicate that certain system customizations, such as system properties containing hardware identifiers, could pose significant security risks. These concerns may represent only the tip of the iceberg. Industry stakeholders must prioritize system security in customization processes by regularly conducting internal audits for vulnerabilities and, more importantly, thoroughly reviewing market apps to uncover potential exploitation methods. Implementing robust authentication methods and continuously reviewing security practices in market will be essential in safeguarding user privacy and maintaining trust.

Limitation. First, our approach used static analysis to examine device-specific behaviors in large-scale real-world apps, focusing on Java code in unpacked apps. This prevented us from addressing packed apps, native code, and dynamically loaded code. Additionally, due to limitations in static analysis tools, some unpacked apps could not be analyzed. To validate system properties with hardware identifiers, we relied on cloud testing platforms. However, it's uncertain if the platform's customizations for cloud testing could affect device security and consequently impact our results.

6 RELATED WORK

API-related Compatibility Issues. API-related compatibility issues have been widely studied. Li et al. [42] developed a static analysis method to identify these issues directly from Android app bytecode, modeling API lifecycles to evaluate potential compatibility problems. Huang et al. [41] examined callback API issues and developed CIDER, a tool that uses static analysis and Callback Control Flow Graphs to detect inconsistencies. Xia et al. [48] presented RAPID, an automated tool for detecting compatibility issues in apps, and employed a learning-based approach to classify these compatibility checks.

Device-related Compatibility Issues. Research on device-related compatibility issues is less comprehensive compared to API-related issues. Zhao et al. [51] developed RepairDroid, a tool designed to automatically address compatibility issues in Android apps, including device-specific ones, using an app patch description language. Wei et al. [46] manually analyzed the source code of 27 open-source apps to identify device-related compatibility issues, proposing a formalism for these issues and developing FicFinder for automated detection. However, FicFinder is limited to open-source apps, which may not undergo the same rigorous device-specific refinements as commercial apps, as noted in previous studies [43]. Wei et al. [47] introduced Pivot, a technique for extracting API-device correlations from Android apps and prioritizing them based on their likelihood of causing compatibility issues. Pivot was validated with over 5,000 apps from Google Play and identified 17 compatibility issues, although only scattered ones corresponded to 10 functionalities.

7 CONCLUSION

We performed an in-depth examination of device-specific behaviors in a broad range of real-world Android apps. Our extensive analysis, covering over 20,000 apps, identified 2,357 apps exhibiting device-specific behaviors. These behaviors were primarily employed for issue fix and adapting to features introduced by manufacturers. Furthermore, we uncovered device-specific behaviors related to privacy. Certain apps leverage unprotected system properties to extract non-resettable identifiers without the need for any permissions, posing a substantial risk to user privacy. Our research not only offers developers valuable insights into device-specific behaviors but also underscores the imperative for enhancing documentation and fortifying the security of customized code. Our findings advocate for a more proactive industry stance in addressing these challenges, ensuring safer and more reliable mobile ecosystem for all users.

ACKNOWLEDGEMENTS

This work was supported by National Key Research and Development Program of China (No.2022YFB3104400), 2023 Special Program for Industrial Base Reconstruction and High-quality Development of Manufacturing Industry (No.0747-2361SCCZA170), the National NSF of China (grants No.62072046), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the Knowledge Innovation Program of Wuhan-Basic Research (2022010801010083), and HUSTCSE-FiberHome Joint Research Center.

REFERENCES

[1] 2020. Open Anonymous Device Identifier (in Chinese). http://www.msa-alliance.cn/col.jsp?id=120.

[2] 2023. Advertising ID. https://support.google.com/googleplay/android-developer/answer/6048248?hl=en.

[3] 2023. Agora Real-Time Voice and Video Engagement. https://www.agora.io/en/.

[4] 2023. Android Debug Bridge (adb). https://developer.android.com/tools/adb.

[5] 2023. Android Forum for Mobile Phones, Tablets, Hardware & App Development. https://xdaforums.com/.

[6] 2023. Android is for everyone. https://www.android.com/everyone/#:~:text=There%20are%20now%20nearly%201%2C300,Android%20Fragmentation%20Visualized%20%2D%20August%202015.&text=Google%20Play%20offers%20over%201,developers%20all%20over%20the%20world..

[7] 2023. An Android library supports badge notification like iOS in Samsung, LG, Sony and HTC launchers. https://github.com/leolin310148/ShortcutBadger.

[8] 2023. APK packing detection tool. https://github.com/MagiCiAn1/APKProtectionSearch.

[9] 2023. App China Android App Market. http://www.appchina.com/.

[10] 2023. App shortcuts overview. https://developer.android.com/develop/ui/views/launch/shortcuts.

[11] 2023. Device-specific Behaviors. https://github.com/security-pride/Device-specific-Behaviors.

[12] 2023. EMAS - One-Stop Application Development Platform Product System. https://www.aliyun.com/product/emas.

[13] 2023. GSMArena.com - mobile phone reviews, news, specs. https://www.gsmarena.com/.

[14] 2023. Huawei AppGallery. https://appgallery.huawei.com/.

[15] 2023. JPush. https://www.jiguang.cn/en/push.

[16] 2023. Learn about foldables. https://developer.android.com/guide/topics/large-screens/learn-about-foldables.

[17] 2023. Markets in AndroZoo. https://androzoo.uni.lu/markets.

[18] 2023. Mi Developer. https://dev.mi.com/platform.

[19] 2023. Mobile OS market share worldwide 2009-2023. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

[20] 2023. One of China's most popular apps has the ability to spy on its users. https://www.cnn.com/2023/04/02/tech/china-pinduoduo-malware-cybersecurity-analysis-intl-hnk/index.html.

[21] 2023. OPPO ColorOS 14 Global Version Officially Starts to Rollout, Featuring Smart and Smooth Experiences. https://www.oppo.com/en/newsroom/press/oppo-coloros-14-global-official-version-rollout/.

[22] 2023. OPPO Developer. https://open.oppomobile.com/.

[23] 2023. Push Kit - Cloud Message Push. https://developer.huawei.com/consumer/en/hms/huawei-pushkit/.

[24] 2023. Recents screen. https://developer.android.com/guide/components/activities/recents.

[25] 2023. Samsung Developers. https://developer.samsung.com/.

[26] 2023. Send notifications across platforms at no-cost - Firebase. https://firebase.google.com/products/cloud-messaging.

[27] 2023. Stack Overflow - Where Developers Learn, Share, & Build Careers. https://stackoverflow.com/.

[28] 2023. Stealthy Sensitive Information Collection from Android-Apps. https://i.blackhat.com/Asia-23/AS-23-Bai-Stealthy-Sensitive-Information-Collection-from-Android-Apps.pdf.

[29] 2023. This library helps bring up the autostart permission manager of a phone to the user so they can add an app to autostart. https://github.com/judemanutd/AutoStarter.

[30] 2023. vivo Funtouch OS 14. https://www.vivo.com/en/funtouch.

[31] 2023. WeTest - one-stop quality open platform officially produced by Tencent. https://wetest.qq.com/.

[32] 2023. Xiaomi Push. https://mipush.global.xiaomi.com/.

[33] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories*. 468–471.

[34] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[35] Jakob Bleier and Martina Lindorfer. 2023. Of Ahead Time: Evaluating Disassembly of Android Apps Compiled to Binary OATs Through the ART. In *Proceedings of the 16th European Workshop on System Security*. 21–29.

[36] Zikan Dong, Hongxuan Liu, Liu Wang, Xiapu Luo, Yao Guo, Guoai Xu, Xusheng Xiao, and Haoyu Wang. 2022. What did you pack in my app? a systematic analysis of commercial Android packers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1430–1440.

[37] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. 2018. Things You May Not Know About Android (Un) Packers: A Systematic Study based on Whole-System Emulation.. In *NDSS*.

[38] Zeinab El-Rewini and Yousra Aafer. 2021. Dissecting residual APIs in custom android ROMs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1598–1611.

[39] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 83–92.

[40] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. 2022. Large-scale security measurements on the android firmware ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 1257–1268.

[41] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.

[42] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.

[43] Pei Liu, Qingxin Xia, Kui Liu, Juncai Guo, Xin Wang, Jin Liu, John Grundy, and Li Li. 2023. Towards automated Android app internationalisation: An exploratory study. *Journal of Systems and Software* 197 (2023), 111559.

[44] Pei Liu, Yanjie Zhao, Mattia Fazzini, Haipeng Cai, John Grundy, and Li Li. 2023. Automatically Detecting Incompatible Android APIs. *ACM Transactions on Software Engineering and Methodology* (2023).

[45] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[46] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.

[47] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning api-device correlations to facilitate android compatibility issue detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 878–888.

[48] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, et al. 2020. How Android developers handle evolution-induced API compatibility issues: a large-scale study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 886–898.

[49] Lei Xue, Hao Zhou, Xiapu Luo, Le Yu, Dinghao Wu, Yajin Zhou, and Xiaobo Ma. 2020. Packergrind: An adaptive unpacking system for android apps. *IEEE Transactions on Software Engineering* 48, 2 (2020), 551–570.

[50] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. 2021. Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4014–4040.

[51] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards automatically repairing compatibility issues in published android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 2142–2153.

[52] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 409–423.