Exploring Covert Third-party Identifiers through External Storage in the Android New Era

Zikan Dong[1]*, Tianming Liu[2,3]*, Jiapeng Deng[3], Haoyu Wang[3]‡, Li Li[4]
Minghui Yang[5], Meng Wang[5], Guosheng Xu[1]‡, Guoai Xu[6]

[1]*Beijing University of Posts and Telecommunications* [2]*Monash University*
[3]*Huazhong University of Science and Technology*
[4]*Beihang University* [5]*OPPO* [6]*Harbin Institute of Technology (Shenzhen)*
* Co-first authors ‡Corresponding author: haoyuwang@hust.edu.cn

Abstract

Third-party tracking plays a vital role in the mobile app ecosystem, which relies on identifiers to gather user data across multiple apps. In the early days of Android, tracking SDKs could effortlessly access non-resettable hardware identifiers for third-party tracking. However, as privacy concerns mounted, Google has progressively restricted device identifier usage through Android system updates. In the new era, tracking SDKs are only allowed to employ user-resettable identifiers which users can also opt out of, prompting SDKs to seek alternative methods for reliable user identification across apps. In this paper, we systematically explore the practice of third-party tracking SDKs covertly storing their own generated identifiers on external storage, thereby circumventing Android's identifier usage restriction and posing a considerable threat to user privacy. We devise an analysis pipeline for an extensive large-scale investigation of this phenomenon, leveraging kernel-level instrumentation and UI testing techniques to automate the recording of app file operations at runtime. Applying our pipeline to 8,000 Android apps, we identified 17 third-party tracking SDKs that store identifiers on external storage. Our analysis reveals that these SDKs employ a range of storage techniques, including hidden files and attaching to existing media files, to make their identifiers more discreet and persistent. We also found that most SDKs lack adequate security measures, compromising the confidentiality and integrity of identifiers and enabling deliberate attacks. Furthermore, we examined the impact of Scoped Storage - Android's latest defense mechanism for external storage on these covert third-party identifiers, and proposed a viable exploit that breaches such a defense mechanism. Our work underscores the need for greater scrutiny of third-party tracking practices and better solutions to safeguard user privacy in the Android ecosystem.

1 Introduction

Third-party tracking refers to the process of collecting data about a user's online behaviors and activities, by an entity distinct from the website or platform being used by the user [44]. In Android, the entity refers to a third-party tracker other than the app and the Android system. These trackers could be advertisers, data brokers, or other service providers that collect user data across multiple apps for various purposes, which generally include targeted advertising that is personalized to users to enhance the likelihood of users engaging with the ad, or analytics that help improve user experience and retention. Although beneficial to app developers and increasing their revenue, third-party tracking also poses a significant threat to user privacy, as it requires profiling users based on their behaviors or interests. In the process of third-party tracking, cross-app user identification plays a crucial role as it allows trackers to connect disparate pieces of information harvested from different apps to a single user for profiling.

Third-party trackers achieve cross-app user identification through user identifiers. In Android, trackers provide app developers with their own SDKs (Software Development Kits) [46] to be integrated into the app to facilitate the acquisition of user identifiers. These identifiers generally include non-resettable hardware device identifiers (e.g., IMEI) and user-resettable identifiers (e.g., Google Advertising ID [12]). To enhance the protection of user privacy, Google has gradually imposed restrictions on the use of device identifiers throughout the development of Android. Initially, apps only needed the necessary permission to obtain hardware identifiers, which would not change even after a system reset. However, due to growing privacy concerns, Google tightened control over hardware identifiers, first mandating user authorization before obtaining hardware identifiers, and then outright prohibiting access to them, while recommending advertising identifiers as an alternative. Compared to hardware identifiers, advertising identifiers are more unstable: firstly, users can manually reset these identifiers at any time; secondly, users can opt out of advertisement personalization, which directly prevents apps from accessing them.

In this new era where the use of hardware identifiers is heavily restricted, third-party trackers are compelled to seek alternative approaches to reliably identify users across different

apps. In our manual analysis of popular third-party tracking SDKs, we have identified cases where tracking SDKs store their own identifiers on Android's external storage. Typically, these SDKs generate an identifier and store it in a specific shared directory location on external storage when it does not already exist on the device. Other apps with the same SDK can then access this identifier to facilitate cross-app user identification. Although apps must obtain the relevant permissions (which require user authorization at runtime) to access external storage, it is common for apps to request such permissions, and users usually do not deny such requests. Moreover, Android's inadequate management of external storage, particularly shared directories, results in numerous files accumulating on external storage, leaving users unable to determine whether deleting these files would affect app functionality or the Android system. Consequently, these generated identifiers may persist in external storage for a long time.

This type of third-party identifier contravenes Google's restrictions on device identifier usage and poses a huge threat to user privacy. Therefore, we intend to conduct an exhaustive, large-scale investigation into such identifiers in real-world scenarios. Our primary goal is to delve into these covert identifier practices, providing an in-depth analysis of their storage, generation, and security measures, as well as their potential implications. We have developed an analysis pipeline to facilitate our large-scale experiment. First, we devise a dynamic analysis framework that leverages kernel-level instrumentation and UI testing techniques to automate the recording of apps' file operations at runtime. Subsequently, we pick out file candidates that are potentially related to third-party identifiers from a collection of file operations recorded across all apps using heuristics. Next, we re-inspect apps that accessed these candidates to locate the code facilitating the file operations, thereby enabling us to attribute the file operations to a specific party. Finally, we perform an in-depth manual analysis to determine if the file candidate is indeed related to third-party identifiers and further delve into the storage, generation, and security aspects of these identifiers.

By adopting our pipeline to 8,000 Android apps, we identified 17 third-party tracking SDKs that store identifiers on external storage. Furthermore, we discovered that these SDKs employ a variety of methods to store identifiers on external storage, including hidden files and attaching to existing media files, in order to make identifiers more covert and persistent. In terms of security, most SDKs lack sufficient protective measures, leaving the confidentiality and integrity of identifiers vulnerable, which opens doors for deliberate attackers to tamper with identifiers and further compromise user privacy, as demonstrated by the two potential attack vectors we proposed in Section 3.2. We also assessed the influence of Scoped Storage, Android's latest defense mechanism for external storage, on these covert identifiers in Section 5.4, and further proposed a viable exploit that can bypass this defense mechanism, thus posing a risk of reliable third-party user identification even after Scoped Storage's enforcement.

In summary, this work offers these major contributions:

- We highlighted the phenomenon of third-party tracking SDKs covertly storing their identifiers in Android's external storage, which significantly threatens user privacy.
- We developed an analysis pipeline to investigate such covert tracking practices extensively on a large scale, and successfully identified 17 third-party tracking SDKs that store their identifiers on external storage.
- We provided an analysis of how these identifiers are stored, generated, and (not) secured.
- We analyzed the implications of these identifiers post Scoped Storage, and further proposed a feasible exploit breaching Scoped Storage to identify users.

2 Background

2.1 Identifier Evolution

During the progression of Android development, the use of identifiers, especially hardware identifiers, has undergone several significant updates to enhance user privacy. However, before delving into the evolution of identifier usage, it is crucial to first recognize that Android's permission management system allows the app and its integrated SDKs to share the same permissions, which means permissions granted to apps also extend to SDKs. To date, this situation persists[1] even though several permission isolation prototype schemes have been proposed [35, 50, 52]. As a result, when referring to permission-related topics, the terms "apps" and "SDKs" are used interchangeably.

Prior to Android 6.0, tracking SDKs could identify users through a variety of hardware identifiers, such as IMEI, IMSI, serial number, and MAC address. These hardware identifiers would remain unchanged even if the user resets the device completely (i.e. factory reset). Additionally, accessing these hardware identifiers was quite easy for apps. Apps only needed to request the READ_PHONE_STATE permission from users upon installation. Users had to accept all permissions displayed or would be unable to install the app. Once granted the permission, the app would have unlimited access to these hardware identifiers which could not be revoked. In addition to hardware identifiers, SDKs could also use ANDROID_ID to identify users. ANDROID_ID is a 64-bit hexadecimal string generated when the device is first set up. ANDROID_ID would change after a factory reset, however, no permission is required to access it.

Starting from Android 6.0 (API level 23, released in 2015), Google has tightened restrictions on the usage of hardware

[1] We further discuss Google's beta program Privacy Sandbox [24] and its SDK Runtime feature [27] in Section 6.3.

identifiers. The required permissions to obtain hardware identifiers are listed as runtime permissions, which require user consent at runtime and can be revoked at any time.

Subsequently, starting from Android 8 (API level 26, released in 2017), ANDROID_ID was changed to have a different value for each app, effectively ending its use for cross-app user identification.

Lastly, starting from Android 10 (API level 29, released in 2019), apps must have privileged permission to access hardware identifiers, a permission level unavailable to third-party apps. Up to this point, Google has imposed strict restrictions on accessing all hardware identifiers and ANDROID_ID. As an alternative, Google recommends apps use advertising identifiers for advertising and analytics purposes. However, users can reset or even opt out of advertising identifiers at any time, making it difficult for third-party trackers such as advertising and analytics companies to achieve reliable user identification using advertising identifiers.

Until January 2023, a significant majority of Android devices (68%) were running Android 10 or a later version [16]. Tracking SDKs are forced to seek alternative approaches for reliable cross-app user identification in the new era.

2.2 External Storage

The storage system is a crucial component of Android, which allows apps to store data persistently. The storage of Android can be classified into two categories: internal storage and external storage. Apps always have access to their own private directory within internal storage (i.e., /data/data/<package name>) and do not require any permission. Moreover, files stored in internal storage can only be accessed by their owner. Therefore, apps generally use internal storage to store data that they do not want other apps to access.

On the other hand, external storage can be divided into two parts, app-specific directories (e.g., "/storage/emulated/0/Android/data/<packagename>") and shared directories. Unlike internal storage, even app-specific directories can easily be accessed by other apps before Android 10 [39]. Therefore, external storage is often used to store non-sensitive files or those meant to be shared.

Prior to Android 10, reading and writing files on external storage only requires the necessary permissions, namely READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE. These two permissions are used so frequently, users tend to grant apps these permissions. Consequently, a lot of apps, and their integrated SDKs, have unlimited access to external storage before Android 10. However, this changes when a new protection mechanism called `Scoped Storage` [26] is introduced in Android 10 (and fully deployed in Android 11 [28]). With Scoped Storage, an app-specific directory can only be accessed by its owner app. On shared directories, an app can only access files it created (no permission required), except under the following conditions: (a) with READ_EXTERNAL_STORAGE[2], apps can read **media** files created by other apps [10]; (b) writing (i.e. modifying) **media** files created by other apps require user consent on a file-by-file basis at runtime [29]; (c) accessing non-media files in shared directories created by other apps require user manual selection via file browsing [11][3]. However, accessing all files in shared directories is still feasible with MANAGE_EXTERNAL_STORAGE, a permission strictly regulated by Google and is granted only to apps with specialized functions such as file management apps [21].

3 Motivation

3.1 Motivating Example

While exploring the Android system, we came across some unusual files stored in hidden directories on external storage. Upon further investigation, we discovered that these files store identifiers generated by the Alibaba Identifier SDK (package name: com.ta.utdid2) that was utilized across different apps. Alibaba is a major Chinese internet company with billions of app downloads to its name. The Alibaba Identifier SDK has a long-standing history, with the earliest information found on search engines dating back to 2015. We will now delve into its identifier generation and management process, which takes place in the `getUtdid` method of the `com.ta.utdid2.device.UTDevice` class.

The Alibaba Identifier SDK employs a four-step process for identifier generation and management, a pattern commonly observed in many other SDKs we examined, as illustrated in Figure 1. This figure also contains a simplified code representation of the `getUtdid` method and highlights the research concerns associated with each step:

Step 1: Initially, the SDK tries to obtain the identifier from two key values in both system settings and `SharedPreference`, with one plaintext and one AES-encrypted version for each. If both attempts fail, the SDK seeks to access the identifier in file `/storage/emulated/0/.UTSystemConfig/Global/Alvin2.xml` and file `/storage/emulated/0/.DataStorage/ContextData.xml` on external storage, again with one plaintext and one encrypted.

Step 2: The SDK checks the validity of the identifier upon each obtainment. The research concerns for this step involve parts of the security aspect of the identifier, namely its integrity, which verifies that the identifier has not been tampered with. However, Alibaba only performs basic format verification, which can only determine if the identifier is damaged but

[2]Starting from Android 13, this permission becomes obsolete and is superseded by media permissions with finer granularity READ_MEDIA_* [15,20].

[3]There are also other functionality-specific exceptions. For instance, with certain media-related permissions and users' explicit consent, apps can write all media files without file-by-file consent [19]. However, discussing these exceptions in detail would be too verbose, and would digress from our research topic. Readers interested in learning more about these exceptions can refer to our citations for further details.

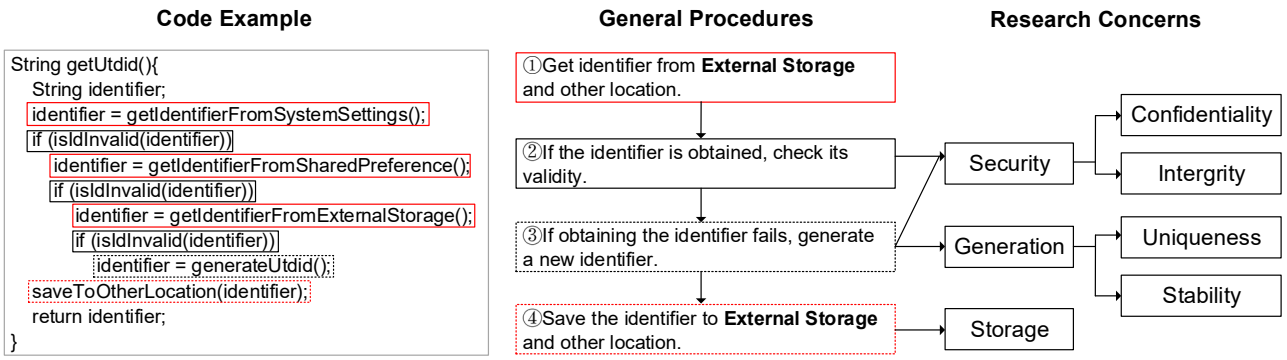


Figure 1: Identifier Generation and Management in Alibaba Identifier SDK: Code Example, Procedures, and Research Concerns

is vulnerable to deliberate attacks elaborated in Section 3.2.

Step 3: If the identifier remains unattainable or invalid from any of the locations mentioned in Step 1, the SDK generates the identifier (based on IMEI or a random value if IMEI cannot be obtained). The research concerns for this step involve: (a) the uniqueness of the generated identifier; (b) whether the same identifier can be generated for a specific device after identifier deletion; (c) the second aspect of the identifier's security, specifically the confidentiality of the identifier. Here, Alibaba ensures uniqueness but cannot generate the same identifier if IMEI cannot be obtained, and only partially considers confidentiality since only one of the two identifiers is encrypted, as mentioned in Step 1.

Step 4: The SDK tries to synchronize the generated identifier across all locations mentioned in Step 1. The research concerns for this step focus on the identifier storage tactics which these tracking SDKs may apply to make their identifier more discreet and persistent. For instance, here, Alibaba stores their identifiers with two backups on external storage, both locations being hidden files users cannot directly access.

Generally, apps do not have access to the system settings as it requires signature-level permission, and `SharedPreference` will be deleted upon the app's uninstallation. Therefore, the primary identifier storage location for the Alibaba Identifier SDK's stable user identification is the two files on external storage, which only require commonly granted storage permissions and can happen stealthily without users' consent, while bypassing Android's system restrictions on identifier usage. This motivating example showcases the importance of thoroughly investigating the research concerns in each step to better understand such identifier practices.

3.2 Potential Threat and Attack Vectors

Tracking users via external storage not only significantly violates user privacy and contravenes relevant identifier policies, but also presents opportunities for deliberate attackers if identifiers are inadequately protected.

Violating Market Policies and Infringing User Privacy: The SDK stores the identifier on external storage, resulting in a long-lasting, relatively stable cross-app identifier accessible to the SDK with only external storage permissions, which stands in contrast to Google's consistent efforts to restrict device identifier usage. Furthermore, this practice also breaches the identifier usage policies established by app stores. For instance, Google Play mandates the use of the Google Advertising ID (when available on a device) over other device identifiers for any advertising purposes [12]. Similarly, Xiaomi Store stipulates that identifiers used for advertising and analytics should be revocable [30]. These identifiers clearly defy this rule, as users cannot manually reset these identifiers or opt out as they can with the Advertising ID. Moreover, such covert user identification can occur without users' consent, and it is almost impossible for an average user to notice such acts. This clearly infringes upon users' privacy choices.

Potential Attacks: Storing identifiers on external storage also opens doors for attackers if the identifiers' confidentiality and integrity are not ensured. We propose two potential attacks.

First, an attacker can use a malicious app on the user's device to modify the identifier on external storage to an identifier carefully crafted by the attacker. After the modification, the SDK will recognize the user's device as the attacker's, leading to the SDK pushing advertisements personalized for the attacker's device to the user. Attackers could thus promote their products and potentially profit from this tactic. The malicious app executing the attack does not require high privileges or dangerous behaviors—only permission to write to external storage and modify the file storing the identifier.

Second, the attacker can read the identifier on external storage, save it on their device, and install an app with the integrated SDK. When using the app, the SDK recognizes the attacker's device as the victim's device since it contains

the victim's identifier, resulting in the attacker receiving advertisements targeted at the victim. By analyzing the content of these ads, the attacker can infer details about the victim's privacy, such as their interests or viewed products. Similar attacks have already been proved feasible both on the web and on the mobile [45, 47].

4 Approach

In order to investigate such covert user identification by third parties through external storage on a large scale, we carefully devised an analysis pipeline, the overview of which is depicted in Figure 2. This pipeline, which takes Android .APK files as inputs, comprises four modules: 1) Dynamic Analysis: in this module, we employ a combination of app automating techniques and kernel-level instrumentation, focusing on one app at a time to capture all file operations during the app's runtime; 2) Candidate Finding: this module is based on the collected file operations from the Dynamic Analysis module, where we pool together file operations derived from a large set of apps to pick out files that could potentially serve as candidates for storing identifiers using heuristics; 3) Candidate Attributing: we further re-inspect the apps that accessed these candidate files to locate the code facilitating the file operation behavior utilizing Frida [18]. This step allows us to attribute potential identifier behavior to a specific component within the app, for instance, a component belonging to a tracking SDK. 4) Manual Analysis: in this final module, we conduct a thorough manual analysis through reverse engineering, to verify if the candidate is indeed related to identifiers, and to further dissect the identifiers in-depth to understand how these identifiers are stored, generated, and (not) secured. It is worth noting that while the first three modules of our approach are largely automated, the threshold selection (detailed in Section 4.2) and the attributing module do require human intervention. We now detail each module as follows.

4.1 Dynamic Analysis

The Dynamic Analysis module facilitates our experiments on large-scale apps and allows us to comprehensively record the file operations of apps at the kernel level. Prior to commencing the analysis, we first initialize our testing devices to simulate the usage patterns of daily-used devices. We then use app automating techniques to traverse apps and record their file operations at runtime for further analysis leveraging instrumentation.

4.1.1 Initialization Settings

To prevent apps from exhibiting abnormal behavior in a completely empty environment (e.g., with no other apps installed, no photos in albums, etc.), we carry out a set of initialization procedures for our dynamic testing environment. Specifically, we begin by installing several widely-used apps on our devices. We then populate the devices with contacts, phone call logs, and SMS message records. Additionally, we take a few photos with the camera and generate some screenshots.

4.1.2 Kernel-Level Instrumentation

To comprehensively capture the file operations performed by the app, we customize the Linux kernel and integrate it into the Android system running on our devices. We monitor file operations at the kernel level since all file operations are eventually handled by the kernel [49]. This approach also allows us to circumvent common instrumentation detection, which generally runs at the framework level [31, 37]. Specifically, we record all system calls invoked by the app that is related to file operations. We achieve this through the following steps:

First, as we aim to record the file operations of the target app only, we flag the threads of the target app with a customized tag at the Zygote process [9] (where all Android apps init) through the system call `prctl` [7], and enable our kernel-level instrumentation only for the flagged threads. This allows us to reduce the impact of other running apps, performance overhead, and log file size.

Second, we configure the kernel to perform system call tracing when a system call (a) originates from a thread containing the aforementioned customized flag, and (b) is related to file operations. There are various types of system calls for different file operation types (e.g., `openat` for opening, `read` for reading, 120 system calls in total for both ARM and ARM64 architectures). The kernel will then call functions `syscall_trace_enter` and `syscall_trace_exit` before and after the execution of the system call respectively, and we can access the relevant information regarding the invoked system call, including parameters and return value, in both functions. We then determine the file operation type based on the type of the system call, and extract the file path from the system call parameters.

Finally, we add a logging module to the kernel that allows us to store the obtained file operations for further analysis. More specifically, we create a file within the `proc` filesystem for logging file operations. `Proc` is a virtual filesystem that provides an interface to access internal data structures in the kernel memory [25]. Initially, we attempted to employ the `printk` function, which is commonly used for logging output in the kernel. However, we discovered that the `printk` function, which stores logs in a circular buffer, is inadequate for handling extensive or high-frequency output [5]. In contrast, our approach of logging within the kernel memory effectively overcomes these limitations, offering enhanced reliability and scalability with respect to the volume of log output.

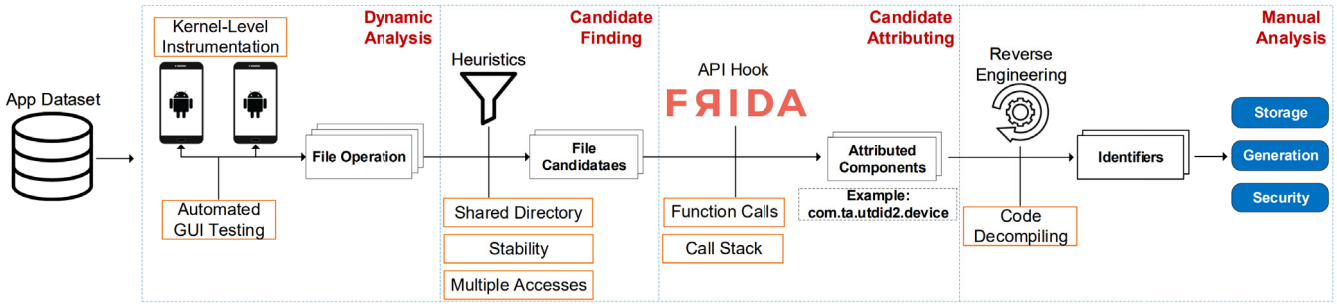


Figure 2: Overview of Our Analysis Pipeline

4.1.3 Automated Testing

Our Dynamic Analysis module aims to record apps' file operations during runtime. To facilitate large-scale analysis, we employ Fastbot [17], an automated GUI testing tool developed by ByteDance. Distinct from other state-of-the-art testing tools, Fastbot is monkey-based (enabling fast action input), and is enhanced by reinforcement learning, therefore outperforming its counterparts in terms of code coverage [42]. During the automated testing, we focus on one app at a time, recording the app's file operations with our instrumentation. After the testing is complete, we uninstall the target app and reboot the device to restore the testing environment for the next app.

4.2 Candidate Finding

After running all the apps in our dataset utilizing our Dynamic Analysis module, we have collected a large number of file operations of these tested apps. It is crucial to automatically identify files on external storage that are potentially associated with identifiers from this large collection of file operations for further analysis. Upon examining the motivating examples and manually probing patterns within the collected file operations, we have derived insights that can help us identify these potential candidates. Finally, we have summarized these insights into the following heuristics to streamline this identification process:

- Files associated with an identifier should be stored in a shared directory on external storage, rather than in an app-specific directory. The file location for storing an identifier should be (a) readily accessible by all apps on the device, and (b) steady enough for consistent user identification. However, files within an app-specific directory are no longer accessible by other apps since Scoped Storage, and more importantly, they are subject to deletion upon the app's uninstallation, making them unsuitable for storing identifiers.

- Files associated with an identifier should not be subject to frequent deletion. Ideal identifiers are device-specific information such as IMEI and serial number, which will remain unchanged even after a factory reset. Files associated with an identifier should also maintain stability, otherwise, it cannot serve its identification purpose.

- If a file is associated with an identifier, it should be accessed by multiple different apps during a large-scale experiment. A tracking SDK needs to be integrated into a large number of apps to facilitate cross-app user identification. As a result, the generated file for storing the identifier is likely to be accessed by a number of different apps in our experiments.

We implement our heuristics as follows. Recall that the collected file operation logs contain both the file operation type and the accessed file path, we first pick out file operations on files in a shared directory on external storage based on the accessed file path. We then eliminate files that are frequently deleted, as indicated by the type of file operations. Next, we count the number of distinct apps each remaining file is accessed by, and utilize this number for our candidate selection. It is vital to note a specific observation from our study: some apps iterate through files on external storage, inevitably inflating the access count for all files. This behavior underscores the need for a heuristic approach when setting the threshold, tailored to the scale of each experiment. For our study, we determined the threshold by sampling files at varying counts. Our rationale for choosing a threshold of 100 is based on the following observations: (a) we found no identifier-related files in sampled files accessed by fewer than 100 apps; (b) our further examination of the top 50 files accessed by fewer than 100 apps yielded none identifier-related; and (c) with the threshold set at 100, we successfully obtained a concise list of only 30 file candidates accessed by 100 or more apps.

4.3 Candidate Attributing

After identifying potential file candidates, we perform a re-inspection of apps to attribute the corresponding file operation to a specific component within the app. To accomplish this,

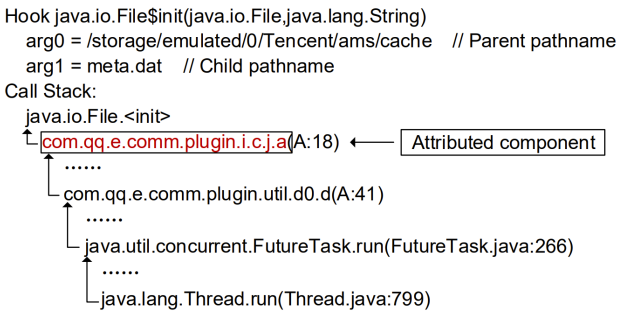


Figure 3: An Example of the Call Stack

we use Frida [18], a dynamic code instrumentation toolkit that allows us to hook common file processing APIs, including Java APIs such as `java.io.File` and `FileInputStream`, as well as the Native API `open` in `libc`. While monitoring calls to these APIs, we also obtain the call stack before the API is executed. The call stack enables us to accurately attribute the file operation to a specific component within the app, as illustrated in Figure 3, where the attributed package name belongs to the Tencent Advertising SDK (package name: com.qq.e). Note that the file processing functions we initially instrumented may not be comprehensive, as it is difficult to enumerate all related APIs. If we fail to attribute a file operation to a specific component, we manually analyze the app to identify the missing file processing API and add it to our hooking list.

In this module, for the apps selected for re-inspection, we re-run the automated testing in the Dynamic Analysis module. During this phase, we utilized Frida for attributing. This decision stemmed from our understanding that directly accessing the caller's package through kernel-level instrumentation (as employed in our preceding module) is infeasible. Consequently, we cannot directly perform the attributing during the app's initial run in the Dynamic Analysis module.

However, it is essential to highlight that Frida's instrumentation, unlike kernel-level instrumentation, can be readily detected by apps, especially those that have undergone app packing. Some developers, in order to enhance app security, may employ app packing techniques that often incorporate anti-dynamic analysis measures, such as running environment detection and anti-debugging techniques [34, 51]. Such measures can effectively thwart our attributing using Frida. Given this context, for each file candidate, we selected one unpacked app that had accessed this file for attributing (and the subsequent manual analysis). This strategy was driven by the strong correlation between the file for identifier storage and the SDK. We deem the attributing successful if we could trace the file operation to a third-party component within the app. If the attributing failed, we'd pivot to another unpacked app to continue attributing.

4.4 Manual Analysis

After attributing the file operations of our selected candidates to a specific component within the app, we perform a thorough manual analysis on the corresponding component. Our analysis consists of two key aspects: first, we ascertain if the file subject indeed contains identifiers; second, we delve deeper into the storage, generation, and security methods applied to these identifiers.

In terms of storage, Android's ineffective management of files on external storage often leads to an accumulation of files in a disorganized manner. Typically, users do not perform thorough cleanups of files on external storage. Consequently, files stored in external storage receive minimal interference from users. Nonetheless, tracking SDKs desire their identifiers to remain as covert and persistent as possible. As such, they may implement protective measures regarding the storage of identifiers. Our manual examination seeks to explore this aspect in greater detail.

For identifier generation, tracking SDKs could effortlessly access unique and stable hardware identifiers via simple API calls in the past, therefore eliminating their concerns over identifier generation. However, in the new era where access to hardware identifiers is restricted, SDKs must generate identifiers of their own. Two primary factors should be considered during identifier generation: uniqueness and stability. Uniqueness is the fundamental requirement for identifiers, as assigning the same identifier to multiple devices would prevent the tracking of a distinct device. Stability in identifier generation means that the same identifier can be generated for a specific device even after the identifier's deletion, app reinstallation, or system reset. While hardware identifiers inherently ensure uniqueness and stability, SDKs must carefully design the identifier generation method to maintain these attributes.

Security should be a vital concern for SDKs when storing identifiers on external storage. This encompasses two key aspects: confidentiality, to prevent unauthorized access or theft of the identifier, and integrity, to ensure the identifier remains unaltered and genuine. Given that files on external storage can be readily accessed by apps with commonly granted permissions, identifiers are at risk of being stolen or even manipulated by other apps on the device or deliberate attackers, which could affect user tracking or even introduce new attack vectors as mentioned in Section 3.2.

Our Manual Analysis module is implemented as follows: since we choose unpacked apps as our analyzing target, we can directly decompile the app using reverse engineering tools. For java code, we use GDA [13], a powerful Dalvik bytecode decompiler, while for native code, we employ IDA Pro [23]. With the rich functionality provided by the decompilers (e.g. string searching, cross-referencing), we start the analysis with the file operation function identified in the Candidate Attributing module, and backtrace the call stack for the caller functions of the file operation function until we

found the file content stored on external storage is consumed (for instance, decrypted and sent to the server). During the backtracing process, we analyze all the code within the control flow, looking for evidence that the contents of the file are identifiers (e.g., usage, function names, log messages). If we establish that the file is indeed related to identifiers, we locate all instances of identifier consumption and further investigate code associated with identifier storage, generation, and security measures. Additionally, in this module, we also filter out potential file operations stemming from external storage scans, and map the package name of the attributed component with its SDK or provider name through a manual lookup of the package name and the associated information.

5 Experimental Results

To facilitate our large-scale investigation, we collect 8,000 Android apps from Google Play and several third-party markets' top listings as inputs to our pipeline. The market split for the apps collected and violating apps with such tracking behaviors are summarised in Table 1. Our experiment is primarily conducted on five Google Pixel3 smartphones running our instrumented Android 9 system, This choice is driven by the introduction of Scoped Storage in Android 10, which could potentially impede the functionality of these third-party tracking SDKs and thereby restrict our ability to fully observe their behaviors. To supplement our study, we further retest a selected group of apps on Android 12 to evaluate the impact of Scoped Storage on these illegitimate tracking behaviors, as detailed in Section 5.4. In our Dynamic Analysis module, we configure Fastbot to perform automated GUI testing on each app for 5 minutes, as we empirically discover that identifier-related file operations are usually executed shortly after app launch, and for scalability concerns.

In total, we collected over 640K file operations for the 8,000 apps in our dataset during the Dynamic Analysis module. Our Candidate Finding module enables us to select 30 file candidates corresponding to 13,735 file operations using heuristics. Through our Attributing and Manual Analysis modules, we verify that 22 of the file candidates contain third-party identifiers. Of the 8 file candidates that did not pass our manual verification, 2 were temporary files for identifier processing[4], while the remaining 6 were unrelated to identifiers, from which we also have some intriguing findings (elaborated in Section 5.5). Therefore, 80% (24/30) of the file candidates are indeed related to identifiers, demonstrating the effectiveness of our heuristics. These 22 confirmed file candidates correspond to 17 tracking SDKs (one SDK might store the identifier in multiple locations, as demonstrated in our motivating example). Among the 8,000 apps subject to our testing, we found 3,337 apps accessed at least one of these identifier files, and 1,947 apps accessed files from multiple tracking SDKs. The name and the number of involved apps for each SDK, along with its storage, generation, and security attributes, are summarized in Table 2. We also list the SDK split for each market and the SDKs' package names in our Appendix. Additionally, we have disclosed the associated apps hosted on Google Play to Google's Trust & Safety Team. They have identified it as an "Abuse Risk". We now detail our findings regarding the storage, generation, and security aspects of the identifiers in the following sections.

5.1 Storage

We empirically find out that tracking SDKs employ a range of storage techniques to ensure their identifiers remain covert and persistent on user devices.

Hidden Files. Out of the 17 SDKs we discovered, 13 of them, including Alibaba, Baidu, and Amap SDKs, store their identifiers in hidden files, which is a relatively common yet effective approach. By default, hidden files are not visible to users. Users need to install third-party file management apps and enable specific settings to access hidden files on external storage. Consequently, this method greatly reduces the possibility of users deleting their identifiers.

Storage Path. Some SDKs store their identifiers in specific directories that could potentially give the impression the file containing the identifier is essential for app functionality.

For example, the ByteDance SDK store its identifier in the directory `/storage/emulated/0/Android/data`, which is the root directory of app-specific directories for all apps. Therefore, users tend to believe that files in this directory are for valid use. In addition, Getui SDK stores the identifier in a file with a "db" suffix in the directory `/storage/emulated/0/libs`, which is a directory commonly used by apps to store databases. After the completion of our experiment, we find hundreds of "db" files in this directory on each of our testing devices. Hiding the identifier file among them can also effectively confuse users.

Multiple Backups. Though users typically do not clean files in shared directories on external storage, there is still a possibility that these files may be lost or damaged. Following the adage, "don't put all your eggs in one basket", 9 SDKs ensure the persistence of their identifiers by backing up the identifier in multiple files, as shown in our motivating example [5].

On top of that, 12 SDKs, such as the Alibaba identifier SDK mentioned in Section 3, go a step further to protect their identifiers by also storing them on internal storage (e.g., using `SharedPreference`), which is accessible only by the host app. After the initial run, the SDK copies the identifiers from

[4]Specifically, the files "Alvin2.xml.bak" and "ContextData.xml.bak" are later renamed to "Alvin2.xml" and "ContextData.xml", respectively, both of which are associated with the Alibaba Identifier SDK, as shown in Table 2.

[5]Certain SDKs from the same organization, namely the two from Baidu and another two from Mob, employ a shared identifier with multiple backups.

Table 1: Distribution of Apps and Violations Across Markets

	# Apps Collected	# Violating Apps	% Violating Apps	# Cumulative Downloads of Violating Apps	# Violating SDKs
Google Play	3,000	102	3.40%	Over 1.9B	16
Huawei	1,000	704	70.40%	Over 84.3B	17
Xiaomi	2,000	1,382	69.10%	Over 11.0B	17
Wandoujia	2,000	1,042	52.10%	Over 529.5M	17
Total	8,000	3,230	40.38%	Over 97.7B	17

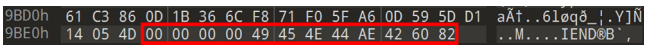
Table 2: Summary of Our Findings on Identifiers of the 17 Tracking SDKs

SDK Name	# Involved Apps	Generation Method	Stability	Uniqueness	Security	Storage Method
Alibaba ID	1,647	System-provided Identifier	○	●	○	/storage/emulated/0/.UTSystemConfig/Global/Alvin2.xml
					◎	/storage/emulated/0/.DataStorage/ContextData.xml
ByteDance AD	1,206	Random	○	●	○	/storage/emulated/0/Android/data/com.snssdk.api.embed/cache/clientudid.dat
Tencent AD	722	Random	○	●	◎	/storage/emulated/0/Tencent/ams/cache/meta.dat
						/storage/emulated/0/Android/data/com.tencent.ams/cache/meta.dat
Baidu Mobstat	542	System-provided Identifier	◎	●	◎	/storage/emulated/0/backups/.SystemConfig/.cuid2
Baidu Map						/storage/emulated/0/backups/.SystemConfig/.cuid
Amap	294	Remote Server	○	/	◎	/storage/emulated/0/backups/.adiu
Mob Share	171	System-provided Identifier	◎	●	○	/storage/emulated/0/Mob/comm/dbs/.duid
Mob SMS					◎	/storage/emulated/0/Android/data/.mn_1006862472
DCloud	173	Random	○	●	○	/storage/emulated/0/.imei.txt
						/storage/emulated/0/.DC4278477faeb9.txt
Umeng*	453	System-provided Identifier	○	●	●	/sdcard/Android/obj/.um/sysid.dat
						/sdcard/Android/data/.um/sysid.dat
Alibaba Quick Login	330	Random	○	●	○	/storage/emulated/0/.pns/.uniqueId/<id>
Kuaishou	265	Remote Server	○	/	◎	/storage/emulated/0/.oukdtft
Getui Push	212	Remote Server	○	/	○	/storage/emulated/0/libs/com.igexin.sdk.deviceId.db
Jiguang	175	Random	○	●	○	/storage/emulated/0/data/.push_deviceid
iFLYTEK	160	System-provided Identifier	○	◎	○	/storage/emulated/0/msc/.2F6E2C5B63F0F83B
Linkedme AD	29	Remote Server	◎	●	○	/storage/emulated/0/.lm_device/.lm_device_id
						/storage/emulated/0/LMDevice/lm_device_id
Shuzilm ID	56	Random	○	●	◎	The earliest created screenshot or photo file

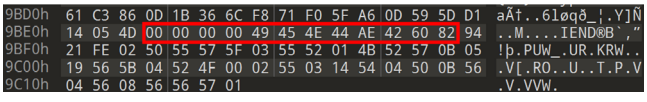
Note: ○ stands for "none", ◎ stands for "partial", ● stands for "complete"
*: Unlike other SDKs, the Umeng SDK generates a new identifier for each app, and only saves the SHA1 hash of the identifier on external storage. Therefore, it does not employ an actual external storage identifier for user tracking. We will discuss this further in Section 5.3.

the external storage to internal storage. When the identifier is used later, the identifier in the internal storage takes priority, and the value of the identifier is synchronized with the file in external storage once retrieved successfully. In our manual analysis, we attempted to delete the file storing the Alibaba identifier on external storage, but quickly discovered that the file was recreated by the Taobao app running in the background, one of the most popular shopping apps in China, and the identifier value in the file remained the same.

Attached to Existing Media Files. Most SDKs store their identifiers separately in individual files, but one SDK employs a more discreet approach. The Shuzilm SDK conceals the identifier within an image file of users. Figure 4 shows the data at the end of a PNG image before and after the SDK embeds its identifier. If a screenshot exists on the user's device, the SDK stores the identifier in the earliest created screenshot; otherwise, the earliest created photo is selected as the storage medium. The SDK stores its identifier by inserting specific data at the end of the image file (either JPG or PNG format). For example, a PNG file starts with an IHDR chunk and ends with an IEND chunk. Adding data after the image end flag typically does not cause the image to display incorrectly. The Shuzilm SDK adds 40 bytes of data to the end of the image file: the first 4 bytes serve as a signature flag, while the remaining 36 bytes store the encrypted identifier.



(a) The End of Image before SDK Inserts Identifier.



(b) The End of Image after SDK Inserts Identifier.

Figure 4: Shuzilm SDK Identifier Storage.

5.2 Generation

In terms of identifier generation, we focus on the uniqueness and stability of identifiers (here, stability means that the same identifier can be generated for a specific device even after the identifier's deletion, app reinstallation, or system reset, as previously mentioned in Section 4.4). Ensuring identifier uniqueness is relatively simple. In our experiment, the majority of SDK identifier generation methods can guarantee uniqueness. However, ensuring the stability of an identifier

is more challenging for SDKs. We observe that no SDK can entirely guarantee identifier stability.

Uniqueness. 5 SDKs employ UUID (i.e., Universally Unique IDentifier) for generating their identifiers. UUID is a 128-bit value generated by the Java class `java.util.UUID` and can guarantee uniqueness across space and time. Some SDKs directly use UUID as the identifier (e.g., Tencent and ByteDance), while others use variants of the UUID. For example, the Alibaba Quick Login SDK uses the MD5 hash of the concatenated UUID, the package name and signature of the app that initially generated the identifier, and the timestamp as its identifier.

```
String generateIdentifier(){
  String systemProvidedIdentifier = "";
  if (APILevel <= 23) {
    systemProvidedIdentifier = getIMEI();
    if (isEmpty(systemProvidedIdentifier)){
      systemProvidedIdentifier = getMacAddress();
      if (isEmpty(systemProvidedIdentifier)){
        systemProvidedIdentifier = getAndroidId();
        if (isEmpty(systemProvidedIdentifier)){
          systemProvidedIdentifier =
              getSerialNumber();
        }
      }
    }
  } else if(APILevel >= 29) {
    systemProvidedIdentifier = getAdvertisingId();
    if (isEmpty(systemProvidedIdentifier)){
      systemProvidedIdentifier = getAndroidId();
      if (isEmpty(systemProvidedIdentifier)){
        systemProvidedIdentifier = getSerialNumber
            ();
        if (isEmpty(systemProvidedIdentifier)){
          systemProvidedIdentifier = getMacAddress
              ();
        }
      }
    }
  } else {
    ......
  }
  String identifier = MD5(systemProvidedIdentifier
      );
  return identifier;
}
```

Figure 5: Code Snippet of Umeng SDK Identifier Generation.

Another 7 SDKs, including Baidu, Umeng, and iFLYTEK, attempt to generate their identifiers using the identifiers provided by the system. For example, the Umeng SDK adopts different identifier generation strategies depending on the system version, as shown in Figure 5. As mentioned in Section 2.1, different types of identifiers are allowed to be accessed on different Android versions. When the Android API level is less than or equal to 23, the Umeng SDK sequentially tries to obtain IMEI, Mac address, ANDROID_ID, and serial number from the system, and uses the first available one to generate its identifier. For API levels between 23 and 29, Umeng changes the acquisition order to IMEI, serial number, ANDROID_ID, and Mac address. When the API level is greater than or equal to 29, the SDK first attempts to obtain the Advertising ID, followed by ANDROID_ID, due to the system restriction on accessing hardware identifiers. Since system-provided identifiers ensure uniqueness, generating identifiers based on them also guarantees uniqueness.

It is worth noting that the iFLYTEK SDK also tries to generate its identifier using system-provided identifiers: it attempts to obtain IMEI and ANDROID_ID. However, if both attempts fail, the SDK simply generates the identifier with a timestamp, potentially compromising the uniqueness.

In addition, 4 SDKs (Kuaishou, Amap, Linkedme, and Getui) do not generate identifiers locally. Instead, they request identifiers directly from remote servers. In this case, the uniqueness of the identifier is determined by the remote server. Lastly, the Shuzilm SDK employs its proprietary algorithm to generate random UUID-style identifiers in the native code.

Stability. As mentioned above, some SDKs use the UUID to generate their identifiers. Since UUID is given a unique value in each generation, these identifiers possess no stability.

Generating identifiers using system-provided identifiers, particularly hardware identifiers, is more likely to maintain stability. However, in the new era where hardware identifiers become inaccessible, their stability is also compromised. From Android 10 onward, ANDROID_ID and the advertising identifier are the only accessible system-provided identifiers, both of which change upon system reset. As ANDROID_ID is given distinct values for different apps, identifiers generated using ANDROID_ID will lose stability if the app that originally generated the identifier is uninstalled. Identifiers generated using the Advertising ID will lose stability when the user manually resets the Advertising ID or the system.

For SDKs that obtain their identifiers from a remote server, their stability is determined by the remote server.

5.3 Security

We proceed to examine the security aspects of these identifiers in terms of confidentiality and integrity. Our findings reveal that almost all (16/17) SDKs do not implement effective protective measures, some even disregard security considerations entirely. Specifically, 8 SDKs entirely neglect the confidentiality of their identifiers, while 7 SDKs overlook the integrity of their identifiers completely. Even among the SDKs that do consider these security aspects, their protective measures are largely ineffective against deliberate attackers.

Confidentiality. 8 SDKs, including Tencent, DCloud, and Getui SDKs, store identifiers on external storage without any encryption. Since no confidentiality protection is in place,

other apps on the device can directly access the identifier, which poses privacy risks and enables new attack vectors as described in Section 3.2.

The remaining SDKs employ encryption for the identifier. However, they all resort to hard-coded keys locally stored within the SDK. This practice compromises the intended confidentiality, especially in the face of determined attackers. For example, the Amap SDK implements AES in CBC mode with a hard-coded static key (the binary format of the ASCII code of a specific string) and an all-zero initialization vector, and further uses string obfuscation techniques to protect the static key and the initialization vector.

Integrity. 7 SDKs (e.g., Alibaba Quick Login, Kuaishou, and Linkedme) do not take identifier integrity into account. After retrieving the identifier on external storage, the SDK directly utilizes it without any input validation. Other apps on the device can directly modify the identifier, and the SDK remains oblivious to any potential tampering.

The other 6 SDKs (e.g., Alibaba, iFLYTEK, and Jiguang SDKs) only perform basic format verification on the identifier and do not examine the actual identifier value. For instance, the ByteDance SDK, which uses the UUID directly as its identifier, only checks the identifier's length and verifies if each character is a number, letter, or hyphen. Moreover, the Amap SDK saves the MD5 hash of a specific string at the beginning of the file where the identifier is stored and checks it when accessing the identifier. In essence, these format verification methods can only to an extent determine if the identifier is damaged but cannot fully ensure integrity. When an attacker alters the identifier value, it can still pass the verification.

```
// Example of file content: {"v":1,"u":"d4c22e74
    -4762-4ad1-afce-3ee2b7525308","t
    ":1667216542434,"m":"
    C9863EB3B042C74A1374EC82F231A869"}
boolean integrityCheck(String fileContent){
  JSONObject fileContentJson = new JSONObject(
      fileContent);
  String fileVersion = fileContentJson.getString("
      v");
  String identifier = fileContentJson.getString("u
      ");
  String timestamp = fileContentJson.getString("t"
      );
  String fileSignature = fileContentJson.getString
      ("m");
  String fileSignatureCalculated = MD5(identifier
      + timestamp + "[a_specific_string]");
  if (fileSignature == fileSignatureCalculated) {
    return true;
  }
  return false;
}
```

Figure 6: Integrity Checking of Tencent SDKs.

Additionally, 3 SDKs (Tencent, Mob share, and Mob SMS SDKs) use hashing algorithms to perform integrity checks on identifiers. We take the Tencent SDK as an example, whose integrity checking method is shown in Figure 6. Tencent stores a JSON string on external storage that contains the file version, identifier, timestamp, and file signature. Upon generating the identifier, the SDK calculates the file signature, which is a hash of the concatenation of the identifier, file version, timestamp, and a specific string, and stores it in a file. When using the identifier, the SDK calculates the file signature in the same way and compares it with the value in the file. If the two signatures match, the SDK assumes the identifier has not been tampered with. While this method can to an extent verify the identifier's integrity, it cannot defend against a deliberate attacker who analyzes the SDK's verification method. A critical weakness of this approach is storing both the signature and identifier together on external storage, enabling the attacker to tamper with the identifier and modify the file signature simultaneously according to the signature calculation method to pass the validation. A more reliable alternative would involve storing the signature in a secure location, such as internal storage. However, it is difficult for SDKs to find a location that is both secure and shareable by other apps that integrate the SDK.

One exception is the Umeng SDK, which ensures both the confidentiality and integrity of the identifier. Upon our manual analysis, we confirm that Umeng generates a new identifier using available system-provided identifiers (as detailed in Section 5.2) for each app. The SDK then stores the identifier's plaintext in each app's internal storage, and only saves the SHA1 hash of the identifier on external storage, potentially for verifying if the device can be identified cross-app. Umeng's approach does not employ an actual external storage identifier for user tracking.

Other Security Issues. In addition to the missing or ineffective identifier confidentiality and integrity protection measures, we found inconsistencies in security measures in Alibaba and Mob SDKs. For example, the Mob SDK stores the identifier in two files on external storage. One of the files (`/storage/emulated/0/.mn_1006862472`) adopts a higher level of security protection (using AES encryption in ECB mode for confidentiality and file signature verification for integrity). However, the other file (i.e., `/storage/emulated/0/Mob/comm/dbs/.duid`) stores the identifier in plaintext and is retrieved by the SDK directly with no integrity check, rendering the protection on the first file ineffective.

5.4 Impact of Scoped Storage

As discussed in Section 2.2, Scoped Storage is introduced in Android 10 (and fully deployed in Android 11 [28]) to restrict the use of external storage. Since accessing non-media files in shared directories created by other apps requires the strictly regulated MANAGE_EXTERNAL_STORAGE or user con-

sent on a file-by-file basis, Scoped Storage should, in theory, prevent SDKs from identifying users through external storage. We thus retest a selected group of apps integrated with these SDKs on Android 12 to assess its impact. As expected, when Scoped Storage is active, the external storage identifiers of all the SDKs we identified become inoperative. However, we find that most SDKs can still track users via external storage under certain conditions: (a) apps targeting Android 10 or lower can opt out of the shared storage restrictions imposed by Scoped Storage, leaving their integrated tracking SDKs unaffected [28]; (b) certain apps from third-party markets (even job-hunting apps and medical apps) significantly abuse the `MANAGE_EXTERNAL_STORAGE` permission, which is intended solely for apps with specialized functions as mentioned in Section 2.2. We further unpack the implications of such tracking practices post Scoped Storage in Section 6.1.

One exception is the ByteDance SDK, which stores identifiers in a folder under the root directory of the app-specific directories. Although this directory does not belong to any specific app, it still becomes inaccessible after Scoped Storage is introduced and cannot be opted out of.

Exploit post Scoped Storage. In summary, Scoped Storage only offers partial protection against these illegitimate identifiers—when an app targets Android 11 or higher and does not request `MANAGE_EXTERNAL_STORAGE`. With some optimism, we can envision a future where opting out of Scoped Storage is disallowed and the `MANAGE_EXTERNAL_STORAGE` usage is rigorously controlled. However, even in that scenario, Scoped Storage still cannot entirely eradicate user identification via external storage. As detailed in Section 5.2, the Shuzilm SDK utilizes the earliest-created image files to conceal its identifier. While this method becomes ineffective post Scoped Storage, as writing to media files not created by the host app requires file-by-file consent, it has enlightened us to propose a new exploit. An adversarial SDK could hide the identifier within a media file generated by its host app with no permission required. Other apps integrated with the same SDK can then access this identifier using commonly granted read permissions. We have developed a demo app that proves the viability of this exploit and reported this to Google's BugHunter Team. They acknowledged and thanked us, but ultimately categorized this exploit as "Won't Fix". We speculate it is because devising a fix for this exploit on system level is very challenging, given the difficulty of implementing controls more rigorous than the present Scoped Storage to an app or SDK (a) writing superfluous data to their self-generated media files, or (b) reading other apps' media files with relevant permissions.

Nevertheless, the Shuzilm method, along with our proposed exploit, represents an adaptation to Android's evolving security measures, indicating an ongoing arms race between both sides. There is also nothing preventing other tracking SDKs from adopting this exploit in the future. The Android system must devise more effective solutions to gain the upper hand in this never-ending struggle.

5.5 Other Findings

In addition to the identifiers on external storage, we have also found that some SDKs share other types of data between apps through external storage.

Sensitive User Data. Some SDKs store sensitive user data on external storage. For example, the Amap SDK stores encrypted detailed device location data (including latitude, longitude, and place names nearby) in a file on external storage. Other apps with the SDK can access user location data (although possibly outdated) by reading this file, even without location permission. In our experiment, we have captured cases where apps integrated with Amap retrieve user location information from this file without location permission, which significantly infringes upon user privacy. Additionally, malicious actors can also gain access to user location data by decrypting this file.

SDK Log Data. Some SDKs export log data to files on external storage. For example, the Meizu push SDK outputs a large amount of logs generated during app execution to external storage. These logs contain server response data and what appears to be a secret key used in network communications. Upon closer inspection, we found that this "key" correlates with a component in the source code named "HttpKeyMgr". This component is subsequently referenced multiple times across different app runs. Storing these logs on external storage is not only superfluous but also poses potential risks to the SDK and user privacy.

6 Discussion

6.1 Implications

By storing identifiers on external storage, these third-party tracking SDKs could circumvent Android's identifier usage restrictions to track users cross-app stealthily, therefore posing significant risks to user privacy.

The latest Android defense mechanism for external storage, i.e., Scoped Storage, should effectively eliminate such illegitimate identifier usage, in theory. However, it falls short in practice. Firstly, this protection is only activated when an app targets Android 11 or higher and does not request the MANAGE_EXTERNAL_STORAGE permission. Although Google Play enforces strict target API level and permission requirements, other third-party markets do not share the same level of scrutiny. For example, SAMSUNG's markets only recently (late 2022) mandated their apps to target Android 8 or higher [8], while HUAWEI and other Chinese app markets require Android 9 [3]. Furthermore, no explicit restrictions on MANAGE_EXTERNAL_STORAGE permission control have been implemented for the two markets, to our knowledge. In addition, Android devices such as smart TVs, tablets, VR devices, and smartwatches running lower system versions are still exposed. Android's fragmentation issues severely un-

dermine the effectiveness of its proposed defense mechanism, making it likely that such practice will persist for a relatively long time.

Moreover, our proposed exploit demonstrates that even in the best-case scenario where all enforcement measures are in place, the latest Android defense mechanism for external storage can still be breached. These concerns call for the development of more robust solutions to safeguard user privacy.

6.2 Chinese SDKs

A notable observation from our study is the predominance of tracking SDKs originating from China, as highlighted in Table 1. While our experimental dataset encompasses 3,000 top-listed apps from Google Play, which undoubtedly utilize major SDKs from other regions, it was primarily the Chinese SDKs that exhibited such tracking behavior. In fact, among the 102 Google Play apps that we discovered with such tracking behavior, the responsible SDKs were still from China. This phenomenon resonates with findings from previous research, such as that by Reardon et al. [49].

From our experiences, Chinese SDKs do tend to be more aggressive in general. This could partially be attributed to the unavailability of Google Play in China. However, while Google's Advertising ID hinges on the Google Play service and is thus inaccessible in China, this factor alone cannot justify the aggressive tracking strategies employed by these Chinese SDKs. Upon our further investigation, we found that China has introduced an alternative: the OAID (Open Anonymous Device Identifier) [6]. Launched in 2019 by the China Mobile Security Alliance and backed by major Chinese smartphone manufacturers like Xiaomi and Huawei, the OAID mirrors Google's Advertising ID functionally, with capabilities for end-users to reset or disable it. Notably, global advertising SDKs such as AppsFlyer [14] and Adjust [2] have integrated support for OAID since late 2019.

Upon scrutinizing the 17 Chinese SDKs from our study, we found that 12 do access the OAID. However, they also employ external storage identifiers, possibly to circumvent OAID's reset or disable features. While the exact motivation remains elusive due to the opaque server-side processing logic of both identifiers, the simultaneous collection of these identifiers alone raises concern. Moreover, the remaining 5 SDKs have yet to adopt OAID, even four years since its debut.

These tracking methodologies also contravene the policies of Chinese app stores. Apart from Xiaomi Store's requirement for identifiers to be revocable as mentioned in Section 3.2, both Xiaomi and Huawei mandate that a new advertising identifier, once reset, should not correlate with the previous identifier or its derived data without explicit user consent [22, 30]. Such a stipulation can be easily violated when an SDK gathers both OAID and external storage identifiers.

Additionally, 16 out of the 17 SDKs neither disclose such external storage tracking in their EULAs nor offer reset or disable features akin to OAID or Google's Advertising ID. We further sampled 50 violating apps and found no disclosure of such tracking practices in the apps' privacy policies.

In conclusion, while regional differences might influence SDK practices, the tracking techniques employed by these Chinese SDKs still raise significant concerns about user privacy. It underscores the need for more transparent practices and robust regulatory oversight to ensure user data protection.

6.3 Mitigation

To counteract the privacy risks posed by third-party tracking SDKs storing identifiers on external storage, we propose the following mitigation suggestions:

Further expanding security requirements for third-party markets. While Google Play enforces strict requirements for apps, such as targeting recent API levels and conducting rigorous permission reviews, third-party markets often do not uphold the same standards. Given that these markets also serve as popular venues for app downloads, it is imperative they align their security requirements with those of Google Play. In early 2019, we have already seen a collective effort by major Chinese app markets to expand the API level requirements up to Android 8.0, a commendable move highlighted by Google [4]. However, subsequent progress has been stagnant over the years. As Android's fragmentation continues to amplify security and privacy concerns, it is vital for Google to take the initiative in expanding these stringent security requirements across more app markets worldwide. This effort could involve enhancing communication between third-party markets, offering technical support, and advocating for a uniform adoption of rigorous security standards. In doing so, we can ensure a consistent application of security measures, thereby reducing the avenues for SDKs to exploit user identifiers.

Permission separation on Android. Although the implementation of Scoped Storage has made strides toward a more secure and organized external storage, our identified exploit has demonstrated that it still remains possible for SDKs to illegitimately identify users through external storage even with the full enforcement of Scoped Storage. One potential solution could involve the adoption of a permission management system on Android to separate SDK permissions from app permissions. Although several permission isolation prototype schemes [35, 50, 52] have been proposed over the years, the current Android system still allows integrated SDKs to share the same permissions as their host apps, thereby enabling them to covertly exploit sensitive permissions requested by the apps. Luckily, in early 2023, Google initiated a beta project called the Privacy Sandbox on Android [24], which they envision as a "multi-year effort". A standout feature of this project is the SDK Runtime [27], which runs app SDKs in a separate process from the app. By doing so, it meticulously governs the permissions and data access rights of SDKs, effec-

tively curbing potential misuse of external storage and other critical system resources.

Collective efforts between end-users and phone vendors/app stores. While the aforementioned mitigation strategies are promising, their realization involves collaborations between Google, third-party markets, and advertising companies that could easily take years. Thus, we advocate for a more immediate mitigation strategy, pivoting on the collaboration between end-users and their phone vendors or app stores. Firstly, app stores can harness our approach to periodically scrutinize their apps, identifying and subsequently addressing tracking malpractices by the responsible SDKs and the apps that integrate them. Secondly, leveraging the identifier file paths located by our approach, phone vendors or app stores can devise tools that exploit the identifier's lack of integrity to continuously reset these files. Such tools can be integrated into market apps or system ROMs to empower users to routinely clean these identifier files.

6.4 Threats to Validity

Several threats to the validity of our study should be considered. First, the Android ecosystem is vast and fragmented, encompassing numerous third-party app stores that do not adhere to the strict guidelines imposed by Google Play. Our analysis includes 8,000 apps, which, although sizable, cannot cover every third-party tracking SDK or strategy due to the sheer scale and complexity of the Android app ecosystem. Also, our pipeline relies on automated GUI testing, which is inherently subject to the limitations of dynamic testing. Therefore, it cannot achieve full code coverage or trigger all relevant functionality within an app. Consequently, we may not obtain a comprehensive view of how each app and its integrated tracking SDKs interact with external storage. Moreover, while our pipeline is largely automated, manual analysis is still required to examine the storage, generation, and security of identifiers, which, to some extent, limits our ability to expand the scope of our experiments and may introduce potential biases in our findings. In addition, despite our pipeline's ability to identify a significant number of covert identifier storage instances, it might not capture every instance or technique. There may be more sophisticated strategies that remain undiscovered. Therefore, our analysis should serve as a foundation for understanding the current landscape and a starting point for further research into third-party tracking SDK practices in the Android ecosystem.

While our heuristics were effective in identifying file candidates associated with identifiers, there is still a possibility of false negatives. Evaluating the recall of our approach is challenging, as this study is the first to investigate this issue on a large scale, and no ground truth dataset is available. However, before our experiment, we found references to three SDKs exhibiting this behavior: Alibaba (from a blog [1]), Salmonads, and Baidu (from Reardon et al. [49]). Our experiment successfully identified two of these SDKs, with Salmonads being the exception due to it being out of service. This, to some extent, demonstrates the recall of our approach and is the best validation we can provide.

Lastly, our dynamic analysis environment is deployed in an AOSP userdebug version of the Android system, which could potentially impact the validity of our results. Although we have made specific modifications (e.g., system properties and system fingerprint) to resemble a release version, it is uncertain whether these adjustments sufficiently evade all detection techniques employed by apps.

6.5 Future Work

Exploring such tracking practices from multi-dimensional perspectives. By examining a set of apps during a specific timeframe, our study captures a snapshot of such tracking behaviors of that period, thus naturally confining the scope of our analysis. It is essential to highlight that the identifier restrictions that led SDK developers to seek workarounds have been in the Android ecosystem for quite some time. To attain a more holistic view, future investigations should examine these tracking methods through various dimensions. This would encompass (a) temporal analysis: delving into the historical trajectory of these tracking methods and SDKs to identify when these techniques initially emerged and observe their progression over time; (b) SDK interrelationships: investigating the intricate dependencies and dynamics among different SDKs to reveal collaborative or potentially competitive nuances within the SDK landscape; (c) emerging practices: as technological landscapes evolve, new tracking methods, for instance, through other covert channels, might surface. Monitoring these developments will ensure our analysis remains current and relevant.

Branching out to other mobile platforms, such as iOS. The iOS system might also be susceptible to such tracking practices. Achieving such tracking would necessitate a shared location accessible by other apps for storing identifiers. While iOS already enforces a sandboxing mechanism to ensure app isolation, crafty methods might find their way around these barriers. For instance, one could employ a method similar to the one we proposed under the inspiration of Shuzilm in Section 5.4. In the iOS system, apps with commonly granted photo access permissions can read any image without users' attention (though, akin to Android post Scoped Storage, they generally cannot write to photos created by others). This opens up a potential avenue for covert tracking on iOS. We plan to further investigate this in our future research.

7 Related Work

Our study investigates how third-party tracking SDKs covertly store identifiers on external storage to bypass Android's identifier restrictions. In this section, we briefly review relevant

literature, including user identifiers and tracking practices, security issues on external storage, and other covert channels in mobile app ecosystems.

User Identifiers and Tracking Practices. Numerous studies have investigated user identifiers and third-party tracking practices in mobile apps. Binns et al. [32] presented an empirical study of the prevalence of third-party trackers on 1 million apps from Google Play. The authors found widespread user tracking in apps but concentrated on a few trackers such as Google and Facebook. Kollnig et al. [38] presented a study of 24K Android and iOS apps, and compared the performance of Android and iOS in terms of user privacy, observing widespread usage of user identifiers in both mobile operating systems. Rahaman et al. [48] proposed a novel taint analysis technique to address inaccurate static analysis resulting from the use of device identifier variants in Android apps, revealing the abuse of identifiers by popular apps and libraries. Leith et al. [40] investigated the data sent to manufacturers by mobile operating systems, finding that device identifiers and other user data were transmitted to back-end servers even when users configured their systems for minimal data sharing. While these studies examined the use of identifiers in user tracking by Android apps and the Android system, they did not explore the application of identifiers beyond those provided by the system.

External Storage Security. Many previous studies have examined the security issues arising from the use of external storage in mobile apps. Liu et al. [41] investigated sensitive user information stored in external storage by apps, and proposed three attacks that exploit the sensitive data to compromise users' privacy. Du et al. [33] systematically analyzed app usage of external storage, discovering that most apps failed to perform valid input validation when accessing sensitive files on external storage, which enabled various attacks with severe impacts on users. Reardon et al. [49] discovered the behavior of unauthorized sensitive information exchange using external storage, and briefly mentioned third-party tracking SDKs storing identifiers (e.g., IMEI) on external storage. However, they did not investigate this issue in depth. These studies primarily focus on high-level sensitive user data stored on external storage without addressing the more basic yet highly sensitive identifiers stored on external storage.

Other Covert Channels. Outside of external storage, there have been some notable investigations into potential covert channels within the Android system. In 2012, Gasior and Yang [36] implemented a network covert channel within an Android app that could potentially allow data leaks. A more recent exploration by Reardon et al. [49] identified various covert channels in Android that were utilized to bypass permission restrictions to access sensitive information. They observed that certain SDKs utilize a range of covert channels, including ioctl (UNIX "input-output control"), UPnP/SSDP discovery protocols, and the ARP table, to obtain the device's MAC address. Furthermore, some SDKs were found to extract geolocation data from photographs by reading the embedded the EXIF metadata. Lyons et al. [43] systematically examined the presence of sensitive information within another covert channel: the Android system log, and found that high-privileged apps could leverage the log to obtain sensitive data such as the WiFi Router's MAC address and user locations. However, the required high privilege is exclusive to pre-installed apps. In summary, while prior studies have revealed sensitive data, such as hardware identifiers, that could be utilized for user tracking, being obtained through covert channels, these channels alone cannot be exploited for the tracking methods highlighted in our study, owing to the fact that they cannot serve as a shared location to store identifiers where other apps can readily access.

In contrast to existing work, our study specifically investigates the practice of third-party tracking SDKs covertly storing identifiers on external storage, examining the techniques they employ and the implications for user privacy. We contribute an extensive large-scale analysis of this phenomenon, which has not been previously explored in the literature.

8 Conclusion

In conclusion, our research has shed light on the covert practices of third-party tracking SDKs in the Android ecosystem, revealing their persistent efforts to bypass Android's identifier usage restrictions by storing identifiers on external storage. Our study not only highlights the need for enhanced defense mechanisms on Android but also underscores the importance of greater scrutiny of third-party tracking practices to protect user privacy. Through our extensive analysis on 8,000 Android apps, we have identified 17 tracking SDKs employing various storage techniques to make their identifiers more discreet and persistent. Our work provides a valuable foundation for understanding the current landscape of third-party tracking SDKs and serves as a starting point for future research. As user privacy continues to be a growing concern in this digital age, it is essential for the research community, industry stakeholders, and policymakers to work together to develop robust solutions and enforce stricter scrutiny to safeguard user privacy in the ever-evolving mobile app ecosystem.

Availability

Our artifact, including the implementation of the first three modules in our approach and all file operations collected in the experiment, is available at https://github.com/security-pride/tpt.

Acknowledgements

We are deeply grateful to our shepherd and the anonymous reviewers for their insightful comments and sugges-

tions. This work was supported in part by the National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (grants No.62072046 and 62302181), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), and the Knowledge Innovation Program of Wuhan-Basic Research.

References

[1] Analysis of alibaba identifier sdk. https://www.dazhuanlan.com/totolinux5/topics/1466752, 2016.

[2] Adjust now supports oaid for tracking in china. https://www.adjust.com/product-updates/adjust-supports-oaid-for-tracking-in-china/, 2019.

[3] Expanding target api level requirements in 2019. https://android-developers.googleblog.com/2019/02/expanding-target-api-level-requirements.html, 2019.

[4] Expanding target api level requirements in 2019. https://android-developers.googleblog.com/2019/02/expanding-target-api-level-requirements.html, 2019.

[5] Debugging by printing. https://elinux.org/Debugging_by_printing, 2020.

[6] Open anonymous device identifier (in chinese). http://www.msa-alliance.cn/col.jsp?id=120, 2020.

[7] prctl(2) — linux manual page. https://man7.org/linux/man-pages/man2/prctl.2.html, 2021.

[8] Android target api level requirements. https://seller.samsungapps.com/notice/getNoticeDetail.as?csNoticeID=0000007234, 2022.

[9] Overview of memory management. https://developer.android.com/topic/performance/memory-overview#:~:text=The%20Zygote%20process%20starts%20when,code%20in%20the%20new%20process., 2022.

[10] Access documents and other files from shared storage. https://developer.android.com/training/data-storage/shared/media#media_store, 2023.

[11] Access documents and other files from shared storage. https://developer.android.com/training/data-storage/shared/documents-files, 2023.

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

[13] an powerful dalvik bytecode decompiler implemented in c++. https://github.com/charles2gan/GDA-android-reversing-Tool, 2023.

[14] Android oaid implementation in the sdk - appsflyer. https://support.appsflyer.com/hc/en-us/articles/360006278797-Android-OAID-implementation-in-the-SDK, 2023.

[15] Behavior changes: Apps targeting android 13 or higher. https://developer.android.com/about/versions/13/behavior-changes-13#granular-media-permissions, 2023.

[16] Distribution dashboard. https://developer.android.com/about/dashboards, 2023.

[17] Fastbot: A model-based testing tool for modeling gui transitions to discover app stability problems. https://github.com/bytedance/Fastbot_Android, 2023.

[18] Frida: A world-class dynamic instrumentation framework. https://frida.re/, 2023.

[19] Manage groups of media files. https://developer.android.com/training/data-storage/shared/media#manage-groups-files, 2023.

[20] Manifest.permission. https://developer.android.com/reference/android/Manifest.permission#READ_EXTERNAL_STORAGE, 2023.

[21] Permitted uses of the all files access permission. https://support.google.com/googleplay/android-developer/answer/10467955?hl=en#zippy=%2Cpermitted-uses-of-the-all-files-access-permission, 2023.

[22] Policies regarding user privacy - huawei appgallery (in chinese). https://developer.huawei.com/consumer/cn/doc/app/50104-07, 2023.

[23] A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro/, 2023.

[24] Privacy sandbox on android. https://developer.android.com/design-for-safety/privacy-sandbox, 2023.

[25] The proc filesystem. https://docs.kernel.org/filesystems/proc.html, 2023.

[26] Scoped storage. https://source.android.com/docs/core/storage/scoped, 2023.

[27] Sdk runtime. https://developer.android.com/design-for-safety/privacy-sandbox/sdk-runtime, 2023.

[28] Storage updates in android 11. https://developer.android.com/about/versions/11/privacy/storage, 2023.

[29] Update other apps' media files. https://developer.android.com/training/data-storage/shared/media#update-other-apps-files, 2023.

[30] Xiaomi developer ecosystem policy (in chinese). https://dev.mi.com/distribute/doc/details?pId=1321, 2023.

[31] Stefano Berlato and Mariano Ceccato. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications*, 52:102463, 2020.

[32] Reuben Binns, Ulrik Lyngs, Max Van Kleek, Jun Zhao, Timothy Libert, and Nigel Shadbolt. Third party tracking in the mobile ecosystem. In *Proceedings of the 10th ACM Conference on Web Science*, pages 23–31, 2018.

[33] Shaoyong Du, Pengxiong Zhu, Jingyu Hua, Zhiyun Qian, Zhao Zhang, Xiaoyu Chen, and Sheng Zhong. An empirical analysis of hazardous uses of android shared storage. *IEEE Transactions on Dependable and Secure Computing*, 18(1):340–355, 2018.

[34] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[35] Jiaojiao Fu, Yangfan Zhou, Huan Liu, Yu Kang, and Xin Wang. Perman: fine-grained permission management for android applications. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 250–259. IEEE, 2017.

[36] Wade Gasior and Li Yang. Exploring covert channel in android platform. In *2012 international conference on cyber security*, pages 173–177. IEEE, 2012.

[37] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225, 2014.

[38] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. Are iphones really better for privacy? a comparative study of ios and android apps. *Proceedings on Privacy Enhancing Technologies*, 2022(2):6–24, 2022.

[39] Yu-Tsung Lee, Haining Chen, and Trent Jaeger. Demystifying android's scoped storage defense. *IEEE Security & Privacy*, 19(5):16–25, 2021.

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

[41] Xiangyu Liu, Wenrui Diao, Zhe Zhou, Zhou Li, and Kehuan Zhang. Gateless treasure: How to get sensitive information from unprotected external storage on android phones. *CoRR, vol. abs/1407.5410*, 2014.

[42] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022.

[43] Allan Lyons, Julien Gamba, Austin Shawaga, Joel Reardon, Juan Tapiador, Serge Egelman, Narseo Vallina-Rodriguez, et al. Log: It's big, it's heavy, it's filled with personal data! measuring the logging of sensitive information in the android ecosystem. In *Usenix Security Symposium*, 2023.

[44] Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *2012 IEEE symposium on security and privacy*, pages 413–427. IEEE, 2012.

[45] Wei Meng, Ren Ding, Simon P Chung, Steven Han, and Wenke Lee. The price of free: Privacy leakage in personalized mobile in-apps ads. In *NDSS*, pages 1–15, 2016.

[46] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W Felten, Prateek Mittal, and Arvind Narayanan. Watching you watch: The tracking ecosystem of over-the-top tv streaming devices. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 131–147, 2019.

[47] ChangSeok Oh, Chris Kanich, Damon McCoy, and Paul Pearce. Cart-ology: Intercepting targeted advertising via ad network identity entanglement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2401–2414, 2022.

[48] Sydur Rahaman, Iulian Neamtiu, and Xin Yin. Algebraic-datatype taint tracking, with applications to understanding android identifier leaks. In *Proceedings*

of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 70–82, 2021.

[49] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX security symposium (USENIX security 19)*, pages 603–620, 2019.

[50] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

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

[52] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 9–18, 2013.

APPENDICES

Table 3: Distribution of SDK Appearances Across Markets

SDK	Google Play	Huawei	Xiaomi	Wandoujia	Total
1	56	387	667	537	1,647
2	5	230	576	395	1,206
3	3	150	352	217	722
4	16	151	233	142	542
5	6	78	122	88	294
6	8	41	65	57	171
7	2	25	70	76	173
8	7	102	177	167	453
9	4	72	149	105	330
10	1	51	128	85	265
11	5	59	82	66	212
12	2	38	74	61	175
13	1	49	67	43	160
14	0	8	10	11	29
15	12	22	11	11	56

Note: 1 – Alibaba ID, 2 – ByteDance AD, 3 – Tencent AD, 4 – Baidu, 5 – Amap, 6 – Mob, 7 – DCloud, 8 – Umeng, 9 – Alibaba Quick Login, 10 – Kuaishou, 11 – Getui Push, 12 – Jiguang, 13 – iFLYTEK, 14 – Linkedme AD, 15 – Shuzilm ID.

Table 4: Package Names of the Tracking SDKs

SDK Name	Package Name of the Attributed Component	SDK Package Name
Alibaba ID	com.ta.utdid2.b.a.d	com.ta.utdid2
Baidu Mobstat	com.baidu.mobstat.bv	com.baidu.mobstat
Baidu Map	com.baidu.b.g	com.baidu.lbsapi
Shuzilm ID	libdu.so (a native library)	cn.shuzilm
Umeng	com.umeng.commonsdk.statistics.idtracking.j	com.umeng
Mob Share	com.mob.tools.utils.ResHelper	cn.sharesdk
Mob SMS	com.mob.tools.utils.DeviceHelper	cn.smssdk
Getui Push	com.getui.gtc.dim.c.a	com.getui.gtc
Amap	com.loc.o.S	com.loc
ByteDance AD	com.bytedance.embedapplog.y	com.bytedance
Tencent AD	com.qq.e.comm.plugin.i.c.j	com.qq.e
Alibaba Login	com.nirvana.tools.logger.storage.FileStorage	com.nirvana.tools
iFLYTEK	com.iflytek.cloud.msc.util.k	com.iflytek
Jiguang	cn.jiguang.ca.c	cn.jiguang
DCloud	io.dcloud.common.util.TelephonyUtil	io.dcloud
Kuaishou	com.yxcorp.kuaishou.addfp.android.a.d	com.yxcorp.kuaishou
Linkedme AD	com.microquation.linkedme.android.util.i	com.microquation.linkedme