Fragile Deliveries: Inconsistencies in Android Parcel and Their Security Consequences

Hongkai Chen[1], Chao Wang[2], Yuqing Yang[3], Jennifer Miller[1], Tiffany Bao[1], Ruoyu Wang[1], Adam Doupé[1], Zhiqiang Lin[2], Yan Shoshitaishvili[1]
[1]Arizona State University, [2]The Ohio State University, [3]CISPA Helmholtz Center for Information Security
{hongkai.chen,jmill,tbao,fishw,doupe,yans}@asu.edu, wang.15147@osu.edu, zlin@cse.ohio-state.edu, yuqing.yang@cispa.de

Abstract

The *Parcel* mechanism is a key component in inter-process communication in Android. However, due to the lack of security considerations, incorrect implementation of the Parcel mechanism can lead to security vulnerabilities. In the past decade, these security vulnerabilities have impacted numerous Android users. In this paper, we identify two major security issues of the Parcel mechanism. First, the reading and writing components are implemented inconsistently in some Parcelable classes, compromising data integrity. Second, malformed Parcels introduce the potential for Denial-of-Service (DoS) attacks on critical apps. We then describe two types of attacks to exploit these two issues: a privilege escalation attack and the Malformed Parcel DoS attack, the latter of which renders phones unusable and prevents users from accessing critical services. To understand the scope of our proposed attacks across the entire Android ecosystem, we perform the first large-scale analysis on 324 Android firmware samples and 10,161 Android apps. Among them, we identify 36 unique data mismatch vulnerabilities and 3,858 apps vulnerable to the DoS attack. We responsibly disclosed our findings to vendors, and 10 of them have been confirmed. Finally, we propose mitigations against the attacks.

CCS Concepts

• **Security and privacy → Mobile platform security**.

Keywords

Android, Parcel Mechanism, Privilege Escalation, DoS

ACM Reference Format:
Hongkai Chen[1], Chao Wang[2], Yuqing Yang[3], Jennifer Miller[1], Tiffany Bao[1], Ruoyu Wang[1], Adam Doupé[1], Zhiqiang Lin[2], Yan Shoshitaishvili[1]. 2026. Fragile Deliveries: Inconsistencies in Android Parcel and Their Security Consequences. In *The 24th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '26), June 21–25, 2026, Cambridge, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3745756.3809193

1 Introduction

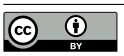
Android dominates the market of mobile operating systems with a 67.46% market share in 2026 [52]. The booming Android ecosystem and large user base make Android security extremely critical. Given the large number of Android developers and vendors, any design flaws will inevitably be misused, consequently compromising user security.

One critical component of Android is the *Parcel* inter-process communication (IPC) mechanism that allows developers to programmatically create custom IPC data transfer protocols. This mechanism includes the Parcel data container and classes that implement the Android Parcelable interface (Parcelable classes) to access the Parcel container. However, due to insufficient security considerations in the Parcel mechanism design, incorrect implementation of Parcelable classes can lead to security vulnerabilities.

In recent years, researchers have reported multiple such vulnerabilities [7], some of which have been exploited in the wild. One example is CVE-2023-20963 [26], which impacted all Android 13 devices and was leveraged by Pinduoduo (a major e-commerce company) to perform privilege escalation attacks on users' Android devices and obtain information such as users' social media accounts [58]. Although a great number of Parcel-related vulnerabilities have been reported, little research has been done to systematically understand these vulnerabilities, determine their security impact, and measure the prevalence of such issues in the Android ecosystem. Ke et al. [7] synthesized a subset of the vulnerabilities into the "Parcel Mismatch Problem", yet their analysis did not cover all types of Android Parcel vulnerabilities, nor did they evaluate the prevalence of the issues.

In this paper, we perform the first large-scale study of the security of the Parcel mechanism. We focus on the cause of Parcel vulnerabilities: *inconsistencies in variable types (Section 4.1) and value constraints (Section 4.2)* of Parcels between receiving and sending or resending (Parcels are often forwarded between apps and privilege levels), driven by errors in Parcel reading and writing functions. An attacker controlling an unprivileged malicious app can leverage these inconsistencies in multiple ways, precisely producing Parcels that trigger errors in victim apps, and even exploiting inconsistencies between Parceling and Unparceling functions to craft Parcels that start out benign (passing initial Framework checks) but become malicious when (re-)unparceled by their final recipient, to execute arbitrary Intents such as granting a malicious app access to system databases and changing passwords by circumventing PIN verification.

We develop an automatic approach to estimate the prevalence of such Parcel inconsistency issues across the Android ecosystem. Using this approach, we scan and analyze 324 Android firmware samples across 9 different Android versions from Android 6 to Android 14, and we discover 716 potential vulnerabilities in 283



This work is licensed under a Creative Commons Attribution 4.0 International License.
MobiSys '26, Cambridge, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2027-7/2026/06
https://doi.org/10.1145/3745756.3809193

firmware samples (87.35% of the analyzed samples), representing 36 unique vulnerabilities. We compare our approach against the state-of-the-art in Android Parcel mismatch detection, showing a 92.9% improvement in identified vulnerabilities.

Additionally, our study revealed a novel end-to-end attack *affecting much of the Android ecosystem* (Section 5), caused by a design flaw regarding assumed consistency between Parcel data and Parcel reading implementation. This flaw allows malicious apps to send Parcels inconsistent with the programmatic Parcel specification at receipt, causing unexpected exceptions and immediate termination in victim apps. Our analysis of 10,161 popular and system apps identified 3,858 apps (37.97%) vulnerable to the DoS attack, including critical apps such as Google Dialer, Messages, Android Auto, Settings, Play Store, and Launcher (the Home Screen). Targeted against these apps, our attack completely disables critical services on user devices, such as making and receiving phone calls (including emergency calls) and texts, leaving them unable to communicate in an emergency. Our attack can also completely disable user devices by DoSing the Home Screen, and prevent users from safely using Android while driving in vehicles, or uninstalling the malicious app. This attack can be repeated stealthily (without revealing the app's identity) and continuously, persisting over reboots and **allowing attackers to render any Android device released in the last 9 years unusable** until recovered via `adb` intervention.

Finally, we propose mitigations against these attacks and integrate our analysis into an Android Studio plugin. We responsibly reported our findings to vendors, and 10 of them have been confirmed. Google has awarded an Honorable Mention prize to recognize our contributions.

Contributions. Our key contributions are:

- We study Android Parcel mechanism security, identifying two types of inconsistency issues in Parcel communication and demonstrating their impact through end-to-end privilege escalation attacks.
- We demonstrate a novel Malformed Parcel attack leading to a complete Denial of Service of Android phones, showing that it can prevent users from accessing critical services (including emergency calls, safe vehicle operation, and even the entire phone UI component).
- We perform the first large-scale analysis on 324 Android firmware samples and 10,161 Android apps to study the prevalence of Parcel security issues and their impact on Android security. Our results show that 87.35% of the firmware samples and 37.97% of the apps are vulnerable to the Parcel security issues.

2 Background and Related Work

Android Parcel mechanism. An Android *Parcel* is a container for a message (data and object references) sent between processes [55]. Android's Parcel APIs support reading and writing of primitive data types, such as `writeByte()` and `readByte()`. Based on these APIs, Parcels store classes that implement the Android-specific *Parcelable* interface [54], consisting of a `readFromParcel()` method for reading data from a Parcel object and a `writeToParcel()` method for writing data to a Parcel object. Developers of Parcelable classes invoke the Parcel API in precise sequences to write relevant fields

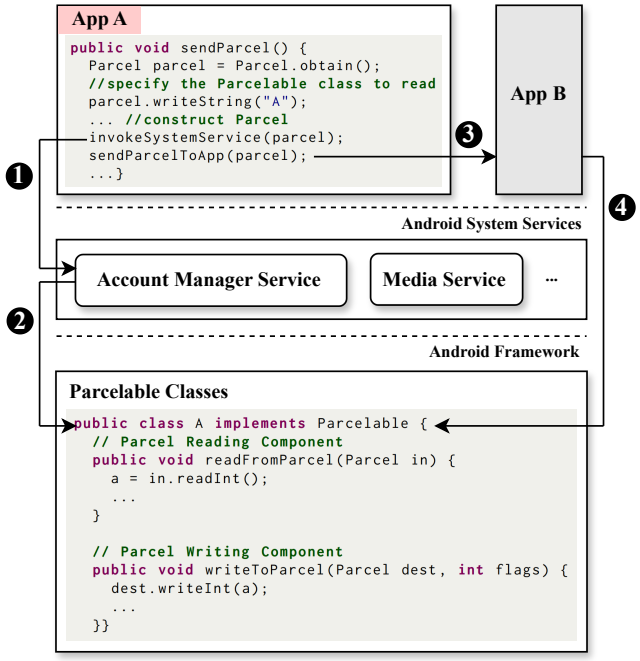


Figure 1: Parcel mechanism communication model. App A invokes system services with the crafted Parcel object in step ❶ and sends the crafted Parcel object to App B in step ❸. System services and App B invoke the corresponding Parcelable class to process the Parcel object in step ❷ and step ❹.

of the object and must, naturally, use the read APIs in the same sequence to read those fields. Parcelable classes, such as class `A` in Figure 1, tend to be implemented by the Android Framework layer. App developers can also define their own Parcelable classes for IPC involving their apps. However, unlike Framework-defined Parcelable classes, which are generally reachable through system IPC interfaces, app-defined Parcelable classes are usually accessible only to limited components within the same app.

Intents and Bundles. Parcel serves as the transport format for Intent and Bundle in Android IPC. An Intent is a message object that represents an action to be performed, for example, starting an activity, invoking a service, or delivering a broadcast [43]. To attach auxiliary data to an Intent, Android commonly uses a Bundle, which is a flexible key-value container supporting primitive values, strings, arrays, and custom objects implementing Parcelable or Serializable [8]. During IPC, Android serializes the Intent and its associated Bundle into a Parcel on the sender side, transmits it through Binder, and reconstructs the data on the receiver side by deserializing the Parcel according to the expected schema.

Utilizing Intents and Bundles, an app (such as App A in Figure 1) can send Parcels both to Android system services (such as the Account Manager Service, which it sends a Parcel to in step ❶) and to other applications (such as App B in step ❸). In both cases, App A can specify the name of the Parcelable class to process (read and write) the Parcel object. Once the recipient receives the Parcel object, it invokes the appropriate `readFromParcel` method, which

```
1  public class ExampleClassA implements Parcelable {
2    // Parcel Reading Component
3    public void readFromParcel(Parcel in) {
4      name = in.readString();
5      id = in.readInt();
6      ct = in.readInt();
7      data = in.createByteArray();
8      ...
9    }
10
11   // Parcel Writing Component
12   public void writeToParcel(Parcel dest, int flags) {
13     dest.writeString(name);
14     // Safe Writing Statement
15     dest.writeInt(id);
16     // Vulnerable Writing Statement
17     dest.writeLong(id);
18     dest.writeInt(ct);
19     dest.writeByteArray(data);
20     ...}}
```

Figure 2: A running example with two inconsistency issues. Line 15 is the correct statement to write `id`. Developers may incorrectly compose line 17 instead, causing a mismatch. A malformed Parcel can also cause errors at line 7.

uses Parcel APIs to process the data (step ❷ for the Android Account Manager and step ❹ for App B).

Parcel mismatch running example. We provide a running example in Figure 2, which is from a real-world vulnerability in the AOSP Framework, to motivate and illustrate the security issues. `ExampleClassA` in the running example contains two issues. The first issue is at line 7, where a malformed Parcel can cause errors. The second issue is at line 17, where developers incorrectly compose the data type, causing a data mismatch illustrated in Figure 3. We will further dissect these security issues in Sections 4 and 5.

Existing work in Android IPC security. The security issues in Android's inter-process communication have been studied over the past decade. For instance, Hay et al. analyzed the unsafe IPC message handling vulnerability due to missing validation [41], and Feng et al. [32] performed a similar missing validation analysis on the Service Binders. Correspondingly, a number of vulnerability detection approaches have been proposed, including static analysis [51], dynamic fuzzing [1, 9, 48, 67, 70], and intent space analysis [47], while DryJIN [13] detects Java-native sensitive information flows in Android's IPC environment. Additionally, multiple works have measured the potential security impacts of vulnerable IPC components [30, 39, 65, 68]. Despite the prolific results in Android IPC security, none of them focus on the security issues in the Parcel mechanism.

Recent works [7, 10, 69] have reported multiple Parcel mismatch vulnerabilities. Among them, Zhan et al. [69] proposed PMDET, an automated detection tool for Parcel mismatch as a short demo paper, but they do not analyze the root cause of the vulnerabilities and their impact. Thus, their work only covers the mismatch issue, while we identify and cover three types of Parcel vulnerabilities. Furthermore, compared to their work, we analyze a far larger dataset of 324 firmware samples and 10,161 apps (compared to their 6 firmware samples). We conduct a comparative evaluation against PMDET and present the results in Section 7.1. In addition, none of these works systematically understand the security of the Parcel

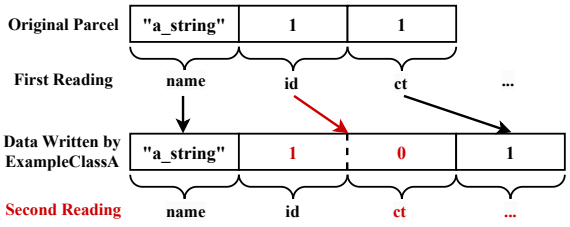


Figure 3: Data mismatches in ExampleClassA

mechanism, measure the prevalence of the security issues in the Android ecosystem, and determine their security impacts. To the best of our knowledge, this paper is the first to explore the Inconsistency in Variable Constraints (Section 4.2) and Inconsistency-Enabled Denial of Service (Section 5), and the first to investigate the security issues in the use of Parcel in apps and systems on a large scale.

Java serialization vulnerabilities. Java serialization vulnerabilities have also been widely examined, including deserialization vulnerability exploitation [59], mitigation and detection [11, 12, 37, 62], and studies on their prevalence. Notably, Peles et al. [56] proposed zero-day vulnerabilities caused by deserialization, and Wu et al. [64] proposed an analysis clustering Android system vulnerabilities, among which a case of a vulnerability caused by inconsistent Parcelable deserialization was discussed.

Compared with previous research in Java serialization vulnerabilities, this paper focuses on Parcel vulnerabilities, which are different from Java's generic serialization vulnerabilities, in the way that a Parcel's format is defined by the developers whereas Java's serialization is standardized. In fact, Parcel is designed as a high-performance, lightweight alternative for Java's serialization. Hence, the security issues stated in this paper do not apply to Java serialization and have not been studied in the previous Java serialization security research.

3 Overview

This section presents an overview of the security problem we study. We begin by describing the threat model and assumptions considered in this paper, including the attacker's capabilities and the assumptions of the targets. We then explain the scope of our study in this work.

3.1 Threat Model and Assumptions

In this paper, we focus on unintended security flaws in the Parcel mechanism design, thus assuming that data flows between the involved parties can be trusted—that is, data flows will not be hijacked or tampered with.

Attackers. We assume that a third-party malicious app is installed on the user's device as the attacker, for example, via an app market. The installed malicious app is a typical third-party app without any permissions or privileges. It has the capability to use unprivileged communication channels, such as an Intent, to initiate IPC with Android system services and other apps by sending Parcels to them. Within this capability, the attacker sends crafted Parcels to exploit inconsistencies in the Parcel mechanism. The attacker's goal is

either to escalate its privileges by bypassing security checks or to trigger DoS conditions in target apps or services by sending corrupted Parcels. Such DoS attacks may be motivated by ransom against specific users or apps, or efforts to degrade the usability of rival apps for competitive advantage.

Targets. We assume that the targets are Android system services in privilege escalation attacks or benign apps in DoS attacks, and that they are exposed to IPC and accept Parcels from unprivileged senders. These targets are not assumed to be malicious or directly compromised. Instead, the attacks succeed because flaws in Parcel-related design or implementation cause the target to interpret crafted Parcel data inconsistently.

3.2 Problem Overview

Inspired by existing Parcel vulnerabilities, this paper aims to present Parcel issues by providing a generic theory explaining the vulnerabilities' essence. Based on this theory, we present variants and justify their security impact by demonstrating associated attacks.

We focus on flaws in Parcelable classes in the Android Framework that may alter data during validation, as these pose meaningful security risks. App-defined Parcelable classes are usually not exported and thus have limited impact, whereas Framework classes can be invoked by any unprivileged app and fit our threat model. Accordingly, we consider only Parcelable classes from the Android Framework in this paper. As our analysis finds an average of 691 Parcelable classes per firmware sample (Section 7.1), the prevalence of Parcelable classes means that even a single mismatch can lead to widespread errors and security risks in Android phones. Third-party Android apps can communicate via IPC in various ways, causing security issues *after* deserializing IPC messages in the past [6]; they are out of scope because we only study security issues caused by the Parcel mechanism.

4 Inconsistencies in Parcelable Classes

In this section, we present two types of inconsistency issues in Parcelable classes, followed by the attacks to exploit the inconsistency issues.

4.1 Inconsistency in Variable Types

At a high level, variable type inconsistency arises when the Parcel writing and reading functions use mismatched types. When a variable's type in a Parcel object is mismatched between the two functions, the Parcel will be different after having been read than it had been before being written, resulting in a time-of-check-time-of-use issue. Using Figure 2 as an example, `ExampleClassA` defines a data structure that can be written to and restored from a Parcel, which contains two key components: (1) The method `readFromParcel()`, which reads a String and two ints from the Parcel object and stores them as variables `name`, `id`, and `ct`, followed by reading a byte array as `data`. (2) The method `writeToParcel()`, which writes the same sequence to a new Parcel object. If developers use line 15 for writing `id`, the data structures read and written by `ExampleClassA` are consistent.

We then consider the scenario where developers incorrectly use `writeLong()` (line 17) for writing `id`. `writeToParcel()` will write variables `name` (a String) and `ct` (an int) correctly. However, it will write the second variable `id` as a long. Since an int is 4 bytes and a long is 8 bytes in Java [45], the Parcel written by `ExampleClassA` differs from the original Parcel provided for reading. `ExampleClassA` may then perform a second read on the written data in a real-world app. For example, as discussed in Section 2, reading a Parcel object to validate the data, writing the data back to a Parcel object, and if the data is legitimate, sending it to the executor, and so on.

To illustrate this inconsistency, Figure 3 visualizes all readings and writings by `ExampleClassA` (omitting `data` for clarity). The first reading is normal. Subsequently, `name`, `id`, and `ct` are written into a new Parcel object. Because line 17 in Figure 2 incorrectly writes `id` as a long, its length becomes 8 bytes: the first 4 bytes represent int 1 and the next 4 bytes int 0. On the second read, `ExampleClassA` reads `name` correctly but interprets the first 4 bytes of the long as `id` and the next 4 bytes as `ct`. As a result, `ct` is incorrectly read, and all subsequent data reads are corrupted.

4.2 Inconsistency in Variable Constraints

Besides type inconsistencies, variables in Parcel objects may also suffer from inconsistencies in their value constraints, especially for variables used for flagging or counting. If the Parcel reading and writing functions have different numeric conditions for these variables, then the variable values might change over repeated reading and writing.

For example, Figure 4a shows `ExampleClassB`, derived from CVE-2023-20963. The `readFromParcel()` method first reads an int as `countNum` followed by an if-else statement. If the condition is true, the program initializes an ArrayList `mUsers` and reads the following data in the Parcel object into the ArrayList. Otherwise, the program sets `mUsers` to null. Then the method reads an int and a String as `score` and `name`. During writing, if `mUsers` is null, the program writes an int `-1`. Otherwise, it writes the size of `mUsers` followed by the ArrayList `mUsers`. Then it writes `score` and `name` as an int and a String to the Parcel object.

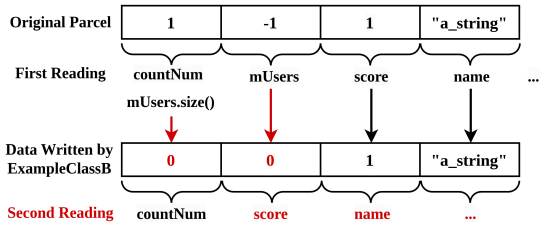
`ExampleClassB` does not seem to contain any inconsistency. However, an inconsistency emerges when it reads a Parcel containing the sequence: int 1, int −1, int 1. As shown in Figure 4b, on the first read, `ExampleClassB` reads the first int 1 as `countNum`. Because `countNum` is greater than zero, it initializes `mUsers` and calls `readParcelableList()`, which interprets the following int -1 as an empty but non-null list [55], setting the ArrayList `mUsers` to empty. The program then reads the next int 1 as `score`.

We now demonstrate how `ExampleClassB` performs writing in this case. Since `mUsers` is an empty list, the program enters the else block (line 20 in Figure 4a) and writes the size of `mUsers`, i.e., int 0, to the new Parcel. It then writes the empty list `mUsers` itself, which again results in writing int 0, followed by writing `score` as int 1. At this point, the first three int values in the Parcel are {0, 0, 1} while the original Parcel data is {1, -1, 1}, resulting in an inconsistency.

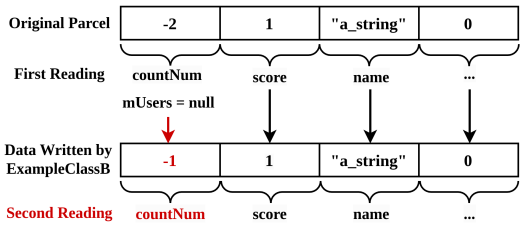
A second read on the rewritten Parcel also leads to inconsistency. Suppose `ExampleClassB` reads the first int 0 as `countNum`. Then the condition at Line 5 does not hold, and the program executes Line 9 and sets `mUsers` to null. At this point, only the first int 0 has been consumed. Subsequently, `ExampleClassB` reads the second int 0 into `score`. Since the correct value of `score` is the third int, the reading of `score` is incorrect, which leads to a data mismatch.

```
1  public class ExampleClassB implements Parcelable {
2    // Parcel Reading Component
3    public void readFromParcel(Parcel in) {
4      countNum = in.readInt();
5      if (countNum > 0) {
6        mUsers = new ArrayList<>(countNum);
7        in.readParcelableList(mUsers, ExampleClassB.class.
               getClassLoader());
8      } else {
9        mUsers = null;
10     }
11     score = in.readInt();
12     name = in.readString();
13     ...
14   }
15
16   // Parcel Writing Component
17   public void writeToParcel(Parcel dest, int flags) {
18     if (mUsers == null) {
19       dest.writeInt(-1);
20     } else {
21       dest.writeInt(mUsers.size());
22       dest.writeParcelableList(mUsers, flags);
23     }
24     dest.writeInt(score);
25     dest.writeString(name);
26     ...}}
```

(a) ExampleClassB



(b) Data mismatches in ExampleClassB



(c) Data changes in ExampleClassB

Figure 4: An example of variable value constraints inconsistency issue in a Parcelable class. The condition in the if-else control flow at line 5 in (a) is flawed, causing the data mismatches in (b) and data changes in (c).

To fix this issue, line 5 in Figure 4a should be changed to `if (countNum >= 0)` by including the case where `countNum` is 0. This will allow `readFromParcel()` to catch the second int 0 during the second read and resolve the mismatch. However, this corner case is subtle and easily overlooked by developers.

4.3 Exploiting Inconsistency Issues

To show the severity of the inconsistency issue, we demonstrate a privilege escalation attack by exploiting `ExampleClassA` in Figure 2, which is derived from a real-world exploit. In step ❶ in Figure 1, App A invokes the Account Manager Service with a crafted Parcel object (see Figure 5). This Parcel object contains two ints 1, followed by an int of the length of the malicious Intent and the malicious Intent itself. The malicious Intent requests the Account Manager Service to grant a higher privilege to App A.

When the Account Manager Service receives the Parcel object, it checks if the Parcel contains a malicious Intent, e.g., an Intent that should not be requested by App A. The essence of this attack is to bypass or evade the check by concealing the malicious Intent. To perform the check, the Account Manager Service invokes the corresponding Parcelable class `ExampleClassA`, which is specified by App A, to read the Parcel object. As illustrated in Figure 5, it reads the first two ints into `id` and `ct`. Then it reads the following data as a byte array (Line 7 in Figure 2). According to the documentation of `createByteArray()`, it reads the first int `length` from the Parcel as the length of the byte array, then reads the malicious Intent data as the content of the byte array. Because the malicious Intent data is hidden within a byte array, Account Manager Service does not recognize it as an executable Intent, determining that the Parcel object sent by App A is safe.

The Account Manager Service incorrectly determines that the Parcel object is safe. It invokes `writeToParcel()` (in Figure 2) to write the data back into a Parcel object for further execution. As shown in Figure 5, due to the mismatch vulnerability (line 17 in Figure 2), it incorrectly writes `id` as a long, and the new Parcel contains four extra bytes of an int 0. The Parcel is then passed to the executor, which reads the data from the Parcel object and executes any executable Intent in the data. The executor invokes `readFromParcel()` in Figure 2 to read the Parcel object. It reads the first four bytes of a long into `id`, and the last four bytes of it into `ct`. Then it reads an int 1 as the length of the byte array, and another int `length` as the byte array itself. At this point, the byte array `data` does not contain the malicious Intent, and the malicious Intent is exposed and gets identified by the executor as an executable Intent. Because the Account Manager Service has already determined that the Parcel object is safe, the executor directly executes the Intent, leading to privilege escalation of App A. We implemented an attacker app and verified that this end-to-end attack can be exploited to bypass permission checks to execute arbitrary Intents, such as

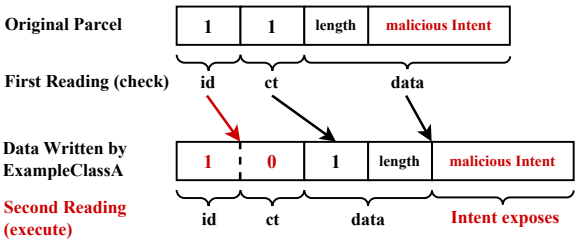


Figure 5: Data mismatch attack process. The attacker crafts the original Parcel object, exploiting the data mismatch vulnerability in Figure 2 to hide the malicious Intent in the first reading, and expose the Intent in the second reading.

```
1  // App A, package name: com.a.a
2  public class MainActivity {
3    public void onCreate(Bundle savedInstanceState) {
4      super.onCreate(savedInstanceState);
5      ...
6      Parcel badParcel = Parcel.obtain();
7      ... // Parcel constructions
8      Bundle bundle = new Bundle();
9      bundle.readFromParcel(badParcel);
10     Intent intent = new Intent();
11     intent.putExtras(bundle);
12     intent.setClassName("com.b.b", "com.b.b.MainActivity");
13     startActivity(intent);
14     ...}}
```

Figure 6: An example app with the component to send a Parcel object to another app.

```
1  // App B, package name: com.b.b
2  public class MainActivity {
3    public void onCreate(Bundle bundle) {
4      super.onCreate(bundle);
5      ...
6      Intent intent = getIntent();
7      Bundle extras = intent.getExtras();
8      Bundle bundle = extras.getBundle("some_key");
9      bundle.keySet();
10     ...}}
```

Figure 7: An example vulnerable app with the component to receive a Bundle from other apps.

granting a malicious app access to system databases and changing passwords by circumventing PIN verification.

5 Inconsistency-Enabled Denial of Service

In analyzing Parcel inconsistencies, we discovered a new variant that results in a high-impact Denial of Service (DoS) attack, which we detail in this section.

5.1 Creating Type Inconsistency in Parcels

The inconsistency issue presented in Section 4 arises during development. Even when Parcelable object implementations are correct, an attacker can still trigger inconsistency by sending a malformed Parcel that does not conform to expected types, causing Android to throw a `BadParcelableException` and crash the victim app. Consider a malicious app `com.a.a` with the component to send a Parcel object (Figure 6) to a victim app `com.b.b` which receives such a Bundle in an Activity class (Figure 7). App `com.b.b` receives an Intent from the app that starts this Activity using the `getIntent()` function at Line 6 in Figure 7. Line 7 gets the Bundle stored in the Intent via `getExtras()`. When App `com.b.b` inspects the Bundle (Line 9), it starts to read the Parcel object. If the Parcel that App `com.a.a` constructed is malformed, Android will throw a `BadParcelableException` and cause a crash in App `com.b.b`.

Furthermore, at the system level, Android lacks comprehensive checks on Parcel data transported in Intents when starting Activities between apps. During app development, developers may omit exception handling when incorporating components that receive Bundles from other apps. This is further compounded by Android's lack of proactive prompts or enforcement regarding exception handling in such components. Thus, if the attacker sends a Parcel in

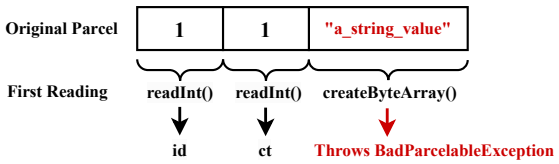


Figure 8: Malformed Parcel DoS attack process. The attacker crafts the original Parcel object with the string value placed intentionally to trigger the `BadParcelableException` in the victim app, leading the victim app to crash.

```
1  public void sendParcel() {
2    Parcel badParcel = Parcel.obtain();
3    ... // construct Parcel
4    Bundle badBundle = new Bundle();
5    Bundle goodBundle = new Bundle();
6    badBundle.readFromParcel(badParcel);
7    Intent intent = new Intent();
8    // bypassing technique
9    intent.putExtras(goodBundle);
10   intent.replaceExtras(badBundle);}
```

Figure 9: Attacker crash bypassing techniques. The attacker can replace a benign Bundle with a Bundle containing the malformed Parcel object to bypass the crash on its app.

the format shown in Figure 8 to a victim app containing the Parcel reading code shown in Figure 2, the victim app will crash upon parsing the Parcel.

5.2 Attack Consolidation

Attacker crash bypassing. The attack workflow is deceptively straightforward. However, during implementation, we found that sending the malformed Parcel object from App A to App B could be challenging. As Figure 6 demonstrates, the malformed Parcel object needs to be placed into a Bundle, which is further placed into an Intent. The Intent is subsequently sent to App B. Normally, App A invokes `putExtras()` to directly write the Bundle object into an Intent. Interestingly, we found that this process invokes `ExampleClassA` to read the Parcel object, which is placed in the Bundle, crashing App A due to a `BadParcelableException`.

This appears to prevent the attack from App A. However, we identify and propose a technique to bypass the issue as illustrated in Figure 9. At Line 9, App A first creates an empty Bundle and places it into the Intent. This process does not cause any crash on App A since `goodBundle` contains no malformed data. Subsequently, App A invokes `replaceExtras()` to replace `goodBundle` in the Intent with `badBundle`, which carries the malformed Parcel object. According to the documentation of `replaceExtras()`, the replacement process does not invoke `ExampleClassA`; instead, it flattens all data in `badBundle` and places it into the Intent. Therefore, this avoids `BadParcelableException` and bypasses the crash in App A.

Continuous and stealthy DoS. In Figure 6, App A invokes `startActivity()` to start and send the malformed Parcel to the victim app. However, due to Android system regulations, App A cannot start the victim app when running in the background. Performing

the attack when App A is in the foreground may reveal the malicious activities, reducing the impact of the attack. We propose a technique to tackle this challenge. Instead of using `startActivity()`, App A can repeatedly invoke `sendBroadcast()` to send the malformed Parcel to the victim app by broadcasting the Intent. The broadcasting event is encapsulated in a service of App A, which can execute in the background and stay hidden. We also manage to set the attack to be launched automatically when the Android device boots, by implementing a `BroadcastReceiver` that listens for the `BOOT_COMPLETED` broadcast, to continue the attack even after users reboot the devices.

5.3 Potential Security Impact

By performing the above attack, the attacker app can delay starting its attack in the background after the user opens it, making the DoS attack stealthy. This attack can crash victim apps several times per second, without graphically revealing the attacker app's identity. The DoS frequency is throttleable by the attacker app as needed. We analyze the apps vulnerable to this attack and present the detailed results in Section 7.2. We performed this attack against a fully up-to-date Android 15 Google Pixel phone, and found that it affects numerous apps, including critical ones, leading to serious security impact.

Freezing device UI. The persistent DoS attack against the Launcher app will freeze the entire device UI and render the phone unusable. An attacker could use this for ransom, e.g., freezing the device until the user pays.

Prohibiting communication. The persistent DoS attack against the Dialer, Messages, and Chrome prevents users from making and receiving calls, reading and responding to texts, or browsing and searching the Internet, leaving them unable to communicate in an emergency.

Disrupting safe in-vehicle use. This attack also prevents users from safely using Android in vehicles by attacking Android Auto and Google Maps. If a user is navigating and driving with Android Auto, this will cause the navigation to disappear suddenly, impacting the driving safety of the user and public safety.

Disabling settings and app management. By persistently DoSing the Settings and Play Store, this attack disables system settings and app management, such as installing and uninstalling apps.

Compromising rival apps. A malicious app can stealthily trigger the attack in the background, repeatedly crashing a rival app as soon as the rival app launches. This may lead users, who are unaware of the root cause, to uninstall the affected apps, thereby boosting the attacker's market share and commercial gains.

Even worse, the attack can be launched automatically when the Android device boots, so it will always occur and freeze the device whenever the device is turned on, and no one can remove the app due to the frozen system.

5.4 Complex Recovery

In practice, users can recover from this attack through safe mode or rescue mode, both of which prevent the attacker app from continuing to run and from crashing the victim app. However, both recovery options are costly. In safe mode, because the attacker often masquerades as a benign third-party app, users still have to identify the malicious app among many installed apps, often by uninstalling apps one by one until the crashes stop. Rescue mode is more direct, but it will erase all data on the phone. Simpler recovery actions are ineffective. For example, restarting the victim app does not resolve the attack, because the attacker app can repeatedly crash the victim app as soon as it is launched.

6 Prevalence Analysis

In this section, we study the prevalence of inconsistency issues by estimating the number of Android firmware and apps that contain such vulnerabilities. Specifically, we aim to answer the two questions:

- How many Android targets contain the inconsistencies in Parcelable classes (as discussed in Section 4)?
- How many Android targets contain the inconsistency-enabled DoS issues (as discussed in Section 5)?

We develop an analysis approach that detects Android targets that potentially contain inconsistencies in Parcelable classes and inconsistency-enabled DoS issues. We will present the design of the approach in this section and the analysis results in Section 7.

6.1 Inconsistencies in Parcelable Classes

As achieving privilege escalation requires that inconsistencies exist in a system receiver, we target Android firmware in this measurement. Our system contains two components: firmware preprocessing and Parcelable class static analysis.

Firmware preprocessing. Given a firmware image, we first unpack it and extract the Android Framework. Appendix D provides more unpacking details for interested readers. We extract all Parcelable classes from the processed Android Framework component. Specifically, for Android 10 through 14, we decompile the framework.jar file using jadx into multiple Java source files. For Android 6, we decompile boot.oat instead. Then we process all Java source files, filter out the Parcelable classes, and export them for further analysis.

Analyzing Parcelable classes. The extracted Parcelable classes will then be analyzed for potential inconsistencies. Given that most static analysis tools like Java PathFinder operate on Java Bytecode [46], their application in our case poses significant challenges. The firmware samples we collect are too large and complicated for bytecode-level analysis. Moreover, decompiling and recompiling the code of each single class is challenging because they require massive Android dependencies. To overcome this, we analyze Parcelable classes by parsing Java source code into Abstract Syntax Trees (ASTs). Our approach prunes a Parcelable class's AST to isolate its reading and writing methods, then identifies basic mismatches by comparing extracted data types and their sequences, using a reference database to account for Android-allowed type equivalents. For more complex classes involving if-else logic, it interprets branch conditions (e.g., null vs. non-null) and checks whether the reading and writing methods apply consistent semantic intervals. Additional analysis details are provided in Appendix E.

6.2 Inconsistency-Enabled DoS

We next analyze Android apps, including critical system apps, to study the prevalence of apps vulnerable to the DoS attack.

Decompiling APKs. Given an Android app, we first decompile the APK file into Java source code and apply a Java AST generator to the source code to generate AST information. We then further prune the AST by removing the components that are unrelated to the app's Broadcast Receivers, such as service classes. Subsequently, we export each Receiver class for further analysis.

Intent receiving feature scanning. To identify whether an app is vulnerable to the DoS attack, we first scan for the usage of receiving an Intent from another app. Specifically, we look for the usage of `getIntent()` function in each Receiver class. Then our analysis checks if the Receiver is accessible by other apps, i.e., it declares `android:exported=true` in its manifest file. If a Receiver of the victim app is accessible by another app and it receives an Intent, the attacker app can start the victim app by invoking `sendBroadcast()` with an Intent.

Bundle usage scanning. We then identify whether an app utilizes the Bundle carried by an Intent. Specifically, we scan for the usage of the pattern `Bundle extras = intent.getExtras()`, which retrieves a Bundle from the Intent. We then identify whether the app uses the Bundle in any format to trigger the invocation of the reading method in the corresponding Parcelable class, e.g., `extras.keySet()` to list all keys in the inner Bundle. If so, we then check whether the app wraps Parcel reading in a try/catch block; if not, we conclude it is vulnerable to the DoS attack.

6.3 Integration for Proactive Detection

We integrate our analysis methodology for privilege escalation prevalence and DoS attack prevalence into an Android Studio plugin to help future app and Android system developers avoid introducing the same security vulnerabilities. This plugin can warn developers when inconsistencies in Parcelable classes and insecure inter-app Bundle usages are detected.

7 Analysis Results

Here, we present the analysis results, including the firmware analysis results for privilege escalation prevalence and the app analysis results for DoS attack prevalence.

7.1 Parcelable Class Inconsistencies

Dataset. To comprehensively study the prevalence and impact of data mismatch vulnerabilities, we perform a longitudinal and cross-vendor analysis using a comprehensive firmware dataset. As previous work has revealed that third-party customization may introduce security issues [29, 40, 57], we include the firmware samples from different vendors in the evaluation. Our firmware dataset spans 9 Android versions from Android 6 to Android 14 (covering 87.5% of the cumulative usage [4]) and 9 major vendors. For each vendor per Android version, we collect 4 firmware samples, totaling 324 firmware images.

Table 1: Aggregate number of vulnerabilities identified by our analysis in each Android version per vendor. For each Android version per vendor, we analyze 4 firmware samples. The aggregate number of vulnerabilities in each Android version is presented at the bottom of the table.

Vendor	V6	V7	V8	V9	V10	V11	V12	V13	V14
Google	11	8	10	0	4	4	12	12	8
Samsung	16	12	14	4	12	12	23	20	12
Xiaomi	11	6	4	8	4	3	12	16	20
OnePlus	10	11	4	4	4	16	17	16	12
Vivo	12	4	1	0	4	3	11	12	21
Motorola	10	4	3	0	3	4	10	14	10
Lenovo	12	6	0	2	4	4	10	10	9
Sony	15	8	5	0	4	4	12	11	12
ZTE	19	6	3	4	6	10	14	16	12
Total	**116**	**65**	**44**	**22**	**45**	**60**	**121**	**127**	**116**

We collected the firmware samples from various sources. Whenever possible, we downloaded them from official vendor websites—for example, all Google Pixel images came from `developers.google.com` [35]. When official sources were unavailable, we relied on reputable third-party sites such as `firmwarefile.com` [33]. Table 8 in Appendix B summarizes all websites and corresponding vendors.

Results. We deploy our analysis on a computer running 64-bit Ubuntu 22.04 on an Intel(R) i7-1280P CPU with 28 logical cores and 32 GiB of RAM. Applying the Parcelable Class Static Analysis (Section 6.1) to all 324 firmware images took a total of 33,048 seconds. Owing to its scalability, the analysis phase takes between 7 and 347 seconds per firmware, with an average of 102 seconds.

Our analysis discovers 716 aggregate data mismatch vulnerabilities across 283 firmware samples—87.35% of all 324 samples. Table 1 summarizes vulnerability counts in each Android version per vendor. Among 81 vendor-version combinations, only 5 contain no vulnerabilities, meaning **93.8%** exhibit at least one issue, demonstrating their widespread prevalence. Android 13 has the highest number of mismatches; counts decline from Android 6 to 9, but rise again from Android 10 to 14 before stabilizing. We include Android 6–8 to show the history of Parcel mismatch and because these versions still account for roughly 8% of active devices—about 312 million users [2, 4].

Observing fewer vulnerabilities in several vendors in V9, we investigate the results of Google V9 and Samsung V9. Google removed 2 of 3 vulnerable classes from V9 and fixed 1; V10 introduced a new vulnerable class. Samsung removed 2 of 4 vulnerable classes from V9, fixed 2; V10 introduced 3 new vulnerable classes. Developers continue to remove and introduce new mismatches, reflecting a lack of understanding of both the root cause and impact. Observing more vulnerabilities in versions after 12, we speculate that this is because the Android Framework has become increasingly complex in recent versions. For example, a typical V9 firmware contains around 500 Parcelable classes, while a typical V13 firmware contains more than 1,500 Parcelable classes, which leads to the occurrence of more vulnerabilities.

Table 2: Vulnerability distributions in AOSP and vendor customization.

	Total	V6	V7	V8	V9	V10	V11	V12	V13	V14
AOSP	615 (85.89%)	97	61	44	19	35	38	105	116	100
Vendor	101 (14.11%)	19	4	0	3	10	22	16	11	16

Our longitudinal analysis shows that these issues are long-standing and recurring design flaws. Although earlier bugs were fixed, similar vulnerabilities reappear in later versions, indicating persistent systemic weaknesses rather than isolated mistakes. This underscores the need for our research.

Table 2 shows the vulnerability distributions in AOSP and vendor customization components. Unsurprisingly, data mismatch vulnerabilities are identified in vendor customization components. Across 9 Android versions from 6 to 14, we identify vulnerabilities introduced by vendors in 8 Android versions, except for Android 8. Of all aggregate vulnerabilities, 85.89% are introduced by AOSP, while 14.11% of the vulnerabilities are introduced by vendors. The results illustrate that while the majority of data mismatch vulnerabilities originate from AOSP, vendors do introduce new vulnerabilities during the customization process.

As we describe in Section 6.1, our analysis produces alerts in the aggregate results when it identifies potential data mismatch vulnerabilities. We process a total of 223,816 classes from the dataset of 324 firmware samples, with an average of 691 classes per firmware sample. Within this set of classes, our analysis produces a total of 716 TPs, while it produces 169 FPs.

Analysis limitations. Our analysis does not support try-catch blocks and switch statements. To quantify the impact, we measured their prevalence in our dataset: only 0.57% of Parcelable classes contain try-catch control flow and 0.51% contain switch statements. Thus, our approach still covers the vast majority of Parcelable classes.

False positives analysis. We manually audited all alerts from our analysis and verified the 169 FPs in about 20 human hours. We inspected the tool logs for each alert for the mismatch pattern, inferred read/write sequence, mismatched constraints, and decompiled file and line locations, then confirmed the reported Parcel read/write sequence by manually inspecting the indicated location. Although our analysis yields 169 FPs, they are easy to verify using the analysis logs, which implies the usability of our approach. All FPs stem from implementation limitations: 64.5% from jadx decompilation inaccuracies, 16.6% from unsupported self-defined types, and 18.9% from control-flow reasoning limitations (as discussed above).

False negatives analysis. To analyze the FNs, we built a benchmark of 13 known data mismatch vulnerabilities and detected 11 of them. We do not identify CVE-2017-0664 [14] and CVE-2021-0928 [24]. For CVE-2021-0928, we perform a case study as shown in Figure 10. Lines 3 to 9 are the vulnerable code, which Google removed in its fix. Line 10 is the code after the fix. The mismatch stems from a try-catch block: a crafted Parcel triggers an exception, causing the program to jump to the catch block and prematurely stop reading, leaving remaining data unconsumed. This leads to

```
1  public class OutputConfiguration implements Parcelable {
2    public void readFromParcel(Parcel source) {
3  -   try {
4  -     OutputConfiguration outputConfiguration = new
         OutputConfiguration(source);
5  -     return outputConfiguration;
6  -   } catch (Exception e) {
7  -     Log.e(TAG, "Exception creating OutputConfiguration from
         parcel", e);
8  -     return null;
9  -   }
10 +   return new OutputConfiguration(source);
11   }
12
13   public void writeToParcel(Parcel dest, int flags) {
14     ...}}
```

Figure 10: Case study of one of the FNs, CVE-2021-0928.

incorrect parsing, but our analysis does not yet support try-catch reasoning. In addition, the other known vulnerability overlooked arises from problems in subsequent data processing. Its Parcel reading and writing methods, which our analysis relies on for detection, are consistent.

Comparison to PMDET. PMDET [69] is a dynamic fuzzing approach, whereas ours is based on static analysis. PMDET supports mismatch detection only on Android 12 and 13, whereas our analysis also covers the DoS attack prevalence analysis and supports all vendor versions from Android 6 through Android 14. Despite our best efforts to reproduce their tool, neither their repository nor their paper (PMDET is a 5-page tool demo paper) documents the necessary instructions to build their Android dependencies. As a result, we were unable to get PMDET running in our environment. Nevertheless, PMDET provided versions of the firmware for their experiment. We carried out a comparative evaluation by collecting and analyzing the same 6 firmware samples reported in their Table 1.

As shown in Table 3, compared to PMDET, our analysis discovered an equal or greater number of mismatches across all firmware samples. In total, our analysis identified **27** mismatches, while PMDET identified 14 mismatches, representing a **92.9%** increase. Additionally, PMDET reported 804 analysis failures on all firmware, due to missing dependencies required for their dynamic, emulator-based approach. Our static analysis avoids such failures. Compared to PMDET, we analyze a far larger dataset of 324 firmware samples and 10,161 apps. A large-scale prevalence analysis across vendors and time is critical in revealing the ongoing seriousness of this issue. As PMDET did not report their False Positive rate, we are unable to compare this aspect of our tool.

From the aggregate results, we identify a total of 36 unique vulnerabilities. Of these, 11 were previously known or reported, which our analysis successfully confirms, and 25 are newly discovered in the wild. We responsibly reported these vulnerabilities to vendors and AOSP respectively, and 9 of them have been confirmed. Table 4 summarizes each vulnerability, including class names, locations, affected Android versions, and available CVE IDs.

Exploitability. Similar to memory corruption bugs, not every Parcel inconsistency vulnerability is exploitable for privilege escalation. The exploitability depends on various factors, including context and previous data checking. Of the 36 unique vulnerabilities we

Table 3: Comparison results of our analysis with PMDET. Our analysis discovered an equal or greater number of mismatches across all firmware. In total, our analysis identified 27 mismatches, while PMDET identified 14 mismatches. PMDET reported 804 analysis failures, while our analysis had none.

Firmware	Android Version	#Mismatches	
		PMDET	Our Analysis
Samsung A217FZHSADWI1	12	1	**6**
Xiaomi V13.0.10.0.SCVCNXM	12	5	5
OnePlus 12.1 H.40	12	2	**4**
Samsung S9010ZCS4CWJ1	13	0	**5**
Xiaomi V14.0.8.0.TLCCNXM	13	3	3
OnePlus 13.1.0.183 F.70	13	3	**4**
Total Mismatches		14	**27**
Total Failures		804	**0**

identified, we verified that 32 are exploitable to trigger a mismatch and bypass Intent checks, enabling arbitrary Intent execution; the remaining four are blocked by sanity or functionality checks during unparceling. For example, one vulnerability becomes unexploitable because the crafted data required to trigger the mismatch is blocked by an unrelated early check in the attack chain.

7.2 DoS Attack Prevalence

Dataset. To comprehensively study the prevalence and impact of the malformed Parcel DoS vulnerability, we assembled a large-scale Android app dataset. Because such attacks can disrupt critical apps and services, we first collected 100 apps from the Tools category of Google Play Store, including security-related apps and popular apps for essential tools. We also collected 61 pre-installed system apps from a Google Pixel 6a device with the latest Android system installed. We then added the top 10,000 apps from the Google Play Store by downloads. Our dataset consists of 10,161 apps in total.

Results. We applied our analysis (Section 6.2) to all 10,161 apps on a 64-bit Ubuntu 24.04 machine with 40 logical cores and 128 GB of RAM. The total runtime of the analysis is 304,422 seconds, averaging about 30 seconds per app.

Out of the 100 apps from the Tools category, we find 46 apps vulnerable to the malformed-Parcel DoS attack. We manually verify each app identified and confirm that it can be crashed on a Google Pixel 6a device with the latest Android system installed. Table 5 lists several vulnerable apps along with their download counts; the full list appears in Table 7 in Appendix A. Many affected apps have massive user bases and critical roles—including Google Keyboard (5 billion+ downloads), major antivirus apps like Kaspersky and Avast, and popular VPNs—meaning successful DoS attacks could have serious security consequences for numerous users.

Among the 61 pre-installed system apps from a Google Pixel 6a device, we identified 18 apps vulnerable to the malformed Parcel DoS attack and verified all on a real-world device (Table 6). Since system apps usually play an important role in an Android device, such as phone services and credential management, performing DoS attacks on these apps also brings serious security risks to users. In addition, from the top 10,000 Play Store apps, we identified 3,794 additional vulnerable apps. In total, 3,858 of 10,161 apps (**37.97%**) are affected. The results indicate that this is a widespread issue with substantial real-world impact (as we discussed in Section 5.3) that could potentially affect a large number of apps on the Google Play Store [36]. We responsibly reported the DoS attack to Google, which has confirmed the issue. We also responsibly disclosed the attack to the developers of all affected apps listed in this paper.

Exploitability. We successfully triggered every manually verified malformed Parcel DoS attack that we attempted to trigger, which breaks the continuous execution flows on Android devices and has been considered a serious security issue in previous academic work [42].

8 Mitigation and Discussion

In this section, we propose potential mitigations for the inconsistency issues and the DoS attack, followed by discussions and ethical considerations.

Inconsistency mitigation. To prevent data mismatches, we propose a simulation mechanism integrated into Parcel. In Figure 1, before App A invokes system services with a Parcel object in step ❶, the simulation mechanism reads the Parcel using the corresponding Parcelable class, rewrites it into a new Parcel, and then reads it again. It compares the two readings: if they match, the simulation mechanism allows further requests by App A, such as sending the Parcel object to system services; if not, it prevents App A from sending the Parcel object. The proposed mitigation mechanism simulates the process of reading, writing, and reading again, such as the checking of malicious Intents as we describe in Section 4.3, and prevents the exploitation of data mismatch vulnerabilities. We reported this mitigation, and Google recently started using the proposed simulation mechanism in some system services [34].

DoS attack mitigation. To prevent the DoS attack, we can adopt the simulation mechanism that we propose for data mismatch mitigation. Specifically, in Figure 1, before App A sends a Parcel object to App B in step ❸, the simulation mechanism reads the content of the Parcel object constructed by App A. It recognizes the corresponding Parcelable class A to process the Parcel object and invokes the reading method. As we illustrated in Section 5, if the Parcel object constructed by App A is malformed, the simulation mechanism will trigger the throwing of a `BadParcelableException`, causing App A to crash, since the simulation process happens before App A sends the malformed Parcel object. This can prevent App A from sending the malformed Parcel object to App B since App A crashes before sending the Parcel object. Regardless, Android and app developers should take care to implement proper exception handling when incorporating components that can potentially receive and utilize Parcel objects from other apps. For instance, using `bundle.keySet()` might trigger the invocation of the reading method of a Parcelable class, and eventually cause a `BadParcelableException`. Android should enforce exception handling when app developers compose such functional components, potentially incorporating this as a feature in Android Studio.

Improvements in Android 13. Although Android 13 introduced safer type-checked APIs [7], they do not effectively mitigate mismatch vulnerabilities. Also, these safer APIs are entirely unrelated to the DoS attacks our paper discusses. They mitigate certain exploit

Table 4: Unique identified data mismatch vulnerabilities. We identify a total of 36 unique vulnerabilities. We confirmed 11 existing CVEs. We responsibly reported all unknown vulnerabilities to the corresponding vendors, and 9 of them have been confirmed.

Vulnerability Class Name	Android Version	Location	CVE (if available)
android.service.gatekeeper.GateKeeperResponse	6, 7, 8	AOSP	CVE-2017-0806 [15]
android.hardware.camera2.params.OutputConfiguration	8	AOSP	CVE-2017-13286 [16]
com.android.internal.widget.VerifyCredentialResponse	6, 7, 8	AOSP	CVE-2017-13287 [17]
android.bluetooth.le.PeriodicAdvertisingReport	8	AOSP	CVE-2017-13288 [18]
android.net.wifi.RttManager	6, 7, 8	AOSP	CVE-2017-13289 [19]
com.android.internal.telephony.DcParamObject	6, 7, 8	AOSP	CVE-2017-13315 [20]
com.android.hotspot2.flow.OSUInfo	8	AOSP	CVE-2018-9431 [21]
android.hardware.location.NanoAppFilter	7, 8, 9	AOSP	CVE-2018-9471 [22]
android.media.MediaPlayer	7, 8, 9	AOSP	CVE-2018-9474 [23]
android.service.gatekeeper.GateKeeperResponse	10, 11, 12	AOSP	CVE-2022-20135 [25]
android.os.WorkSource	11, 12, 13	AOSP	CVE-2023-20963 [26]
nubia.net.wifi.WifiRssiInfo	6	ZTE	-
android.telecom.ParcelableConference	6	Motorola, Lenovo, Sony, ZTE	-
com.mediatek.gba.NafSessionKey	6	Mediatek	-
android.net.DhcpResults	7	Samsung	-
android.hardware.location.NanoAppInstanceInfo	7, 8	AOSP	-
android.hardware.camera2.params.OutputConfiguration	9	Lenovo	-
android.os.sprdpower.AppPowerSaveConfig	10, 11	ZTE	-
android.hardware.camera2.params.OutputConfiguration	9, 10, 11	ZTE	-
android.bluetooth.le.AdvertiseData	9, 12	AOSP	-
com.aiunit.aon.utils.core.FaceInfo	11, 12	OnePlus	-
com.android.internal.telephony.OperatorInfo	11, 12	OnePlus	-
android.content.pm.SemSuspendDialogInfo	10, 11, 12	Samsung	-
com.samsung.android.cocktailbar.FeedsInfo	6, 10, 11, 12	Samsung	-
android.hardware.fingerprint.EngineeringInfo	12	OnePlus	-
android.bluetooth.BluetoothLeBroadcastReceiveState	13, 14	AOSP	-
android.telephony.data.DataProfile	13	Samsung	-
android.hardware.camera2.CaptureRequest	9, 12, 13, 14	AOSP	-
android.hardware.biometrics.SensorLocationInternal	12, 13, 14	AOSP	-
com.samsung.android.knox.util.SemCertByte	12, 13, 14	Samsung	-
android.bluetooth.BluetoothCodecConfig	13, 14	Xiaomi	-
com.miui.enterprise.signature.EnterpriseCer	14	Xiaomi	-
android.bluetooth.BluetoothLeBroadcastSettings	14	AOSP	-
android.graphics.HDRRegion	14	AOSP	-
android.media.tv.TvRecordingInfo	14	AOSP	-
android.os.unisocpower.AppPowerSaveConfig	14	Vivo, Motorola	-

paths but do not eliminate the underlying class of mismatches. Consequently, CVE-2024-49721 [27], a Parcel mismatch-caused Intent check bypass, was still discovered after Android 13's improvements. Moreover, the lack of backward compatibility of these improvements poses significant risks for OEMs. For example, Huawei's latest EMUI (based on Android 12) was found vulnerable to CVE-2025-31175 [28], a classic Parcel mismatch issue. Therefore, Android 13's improvements do not fundamentally resolve the problems analyzed in this work.

Emerging mobile operating systems. In recent years, Android vendors have initiated efforts to implement their own mobile operating systems (OSes). In 2024, Huawei released HarmonyOS NEXT, the successor to their Android-based HarmonyOS, but now containing no Android code and no longer supporting Android apps [38]. HarmonyOS adopts a new language, ArkTS, a customized TypeScript for HarmonyOS app development. We perform a preliminary exploration of the Parcel mechanism in HarmonyOS and discover that although it is a reimplementation of mobile OS and is independent from Google's codebase, it still has a similar Parcel mechanism. We successfully implement a Parcelable class, as shown in Figure 11 in Appendix C, in our testing HarmonyOS app. Similar to Android, the Parcelable class contains two functions, `marshalling()` and `unmarshalling()`, which are equivalent to `writeToParcel()` and

Table 5: A subset of vulnerable apps to the malformed Parcel DoS attack in the Tools category on Google Play Store. DLs: Downloads.

App Package Name	App Name	DLs
com.google.android.inputmethod.latin	Gboard - Google Keyboard	5B+
com.google.android.apps.translate	Google Translate	1B+
com.google.ar.lens	Google Lens	1B+
com.lenovo.anyshare.gps	SHAREit	1B+
com.avast.android.mobilesecurity	Avast Antivirus & Security	100M+
com.kms.free	Kaspersky: VPN & Antivirus	100M+
com.lookout	Lookout Life Mobile Security	100M+
com.antivirus	AVG AntiVirus & Security	100M+
com.google.android.apps.adm	Google Find My Device	100M+
com.google.android.apps.authenticator2	Google Authenticator	100M+
com.mate.vpn	XY VPN	100M+
com.piriform.ccleaner	CCleaner – Phone Cleaner	100M+
com.expressvpn.vpn	ExpressVPN	50M+
com.symantec.mobilesecurity	Norton360 Virus Scanner	50M+

Table 6: Vulnerable pre-installed system apps to the malformed Parcel DoS attack from a Google Pixel 6a device.

App Package Name	App Name
com.google.android.dialer	Phone (Dialer)
com.google.android.apps.messaging	Messages
com.google.android.projection.gearhead	Android Auto
com.google.android.apps.nexuslauncher	Pixel Launcher
com.android.settings	Settings
com.android.vending	Google Play Store
com.google.android.apps.maps	Google Maps
com.android.chrome	Chrome
com.google.android.googlequicksearchbox	Google Search
com.android.certinstaller	Certificate Installer
com.android.providers.contacts	Contacts Storage
com.android.credentialmanager	Credential Manager
com.android.DeviceAsWebcam	Webcam Service
com.android.managedprovisioning	Work Setup
android.process.media	MTP Host
com.android.musicfx	MusicFX
com.android.server.telecom	Phone Calls
com.android.phone	Phone Services

`readFromParcel()` in Android. We implement the class asymmetrically to read an int and write a long and trigger a data mismatch within our test app.

In addition to HarmonyOS, Xiaomi and Vivo have also started implementing self-developed mobile OSes recently [63, 66]. While this paper focuses on the security of Android's Parcel mechanism, these growing trends suggest that our approaches could be extended in future research to Parcel-like mechanisms on other emerging mobile OSes.

Ethical considerations. Upon discovery, we promptly and responsibly disclosed all identified data mismatch vulnerabilities in Table 4 to Google and corresponding vendors, and 9 of them have been confirmed. We also promptly disclosed the Malformed Parcel DoS attack to Google. Google accepted our vulnerability report and awarded an Honorable Mention to recognize our contributions. We are actively working with them to fix the issues. As shown in Section 7.2, we manually verified and crashed 46 apps in the Tools category using the Malformed Parcel DoS attack, and we promptly disclosed these issues to the respective developers. Our research does not involve any experiments that could harm individuals.

9 Conclusion

In this paper, we present inconsistency issues between reading and writing Android Parcels. We show that such issues lead to end-to-end privilege escalation and DoS attacks, and we measure the prevalence of these attacks in Android ecosystems with 324 Android firmware samples and 10,161 Android apps. We identify 36 unique inconsistency issues and 3,858 apps vulnerable to DoS attacks, which reveal the severity of the presented issues. Finally, we propose mitigations against these attacks.

Acknowledgments

We would like to thank our anonymous shepherd and reviewers for their valuable feedback. Further, we thank Kyle Zeng for his assistance with PoC testing. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-22-C-4026, Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074, and the Department of the Interior (DOI) under Grant No. D22AP00145-00. Additionally, we acknowledge the generous support of the US Department of Defense (DOD). The authors at The Ohio State University were supported in part by the National Science Foundation (NSF) under Grant No. CNS-2112471. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, NIWC Pacific, ARPA-H, DOI, DOD, or NSF.

References

[1] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android smarttvs vulnerability discovery via log-guided fuzzing. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2759–2776. USENIX Association, 2021.

[2] Android vs iOS: Mobile Operating System market share statistics (Updated 2025). https://www.appmysite.com/blog/android-vs-ios-mobile-operating-system-market-share-statistics-you-must-know/. Accessed: April 2026.

[3] anyxperia_dumper. https://github.com/munjeni/anyxperia_dumper. Accessed: April 2026.

[4] Android versions, SDK/API levels, version codes, codenames, and cumulative usage. https://apilevels.com/. Accessed: April 2026.

[5] smali. https://github.com/JesusFreke/smali. Accessed: April 2026.

[6] Hey your parcel looks bad - Fuzzing and Exploiting parcelization vulnerabilities in Android. https://www.blackhat.com/docs/asia-16/materials/asia-16-He-Hey-Your-Parcel-Looks-Bad-Fuzzing-And-Exploiting-Parcelization-Vulnerabilities-In-Android.pdf. Accessed: April 2026.

[7] Android parcels: the bad, the good and the better - Introducing Android's Safer Parcel. https://i.blackhat.com/EU-22/Wednesday-Briefings/EU-22-Ke-Android-Parcels-Introducing-Android-Safer-Parcel.pdf. Accessed: April 2026.

[8] Bundle. https://developer.android.com/reference/android/os/Bundle. Accessed: April 2026.

[9] Chen Cao, Neng Gao, Peng Liu, and Ji Xiang. Towards analyzing the input validation vulnerabilities associated with android system services. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 361–370. ACM, 2015.

[10] Sheng Cao, Hao Zhou, Songzhou Shi, Yanjie Zhao, and Haoyu Wang. Parcel mismatch demystified: Addressing a decade-old security challenge in android. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and*

Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025, pages 2683–2698. ACM, 2025.

[11] Bofei Chen, Lei Zhang, Xinyou Huang, Yinzhi Cao, Keke Lian, Yuan Zhang, and Min Yang. Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 3961–3978. IEEE, 2024.

[12] Xingchen Chen, Baizhu Wang, Ze Jin, Yun Feng, Xianglong Li, Xincheng Feng, and Qixu Liu. Tabby: Automated gadget chain detection for java deserialization vulnerabilities. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*, pages 179–192. IEEE, 2023.

[13] Minseong Choi, Yubin Im, Steven Y. Ko, Yonghwi Kwon, Yuseok Jeon, and Hae-hyun Cho. Dryjin: Detecting information leaks in android applications. In Nikolaos Pitropakis, Sokratis K. Katsikas, Steven Furnell, and Konstantinos Markantonakis, editors, *ICT Systems Security and Privacy Protection - 39th IFIP International Conference, SEC 2024, Edinburgh, UK, June 12-14, 2024, Proceedings*, IFIP Advances in Information and Communication Technology, pages 76–90. Springer, 2024.

[14] NVD - CVE-2017-0664. https://nvd.nist.gov/vuln/detail/CVE-2017-0664. Accessed: April 2026.

[15] NVD - CVE-2017-0806. https://nvd.nist.gov/vuln/detail/CVE-2017-0806. Accessed: April 2026.

[16] NVD - CVE-2017-13286. https://nvd.nist.gov/vuln/detail/CVE-2017-13286. Accessed: April 2026.

[17] NVD - CVE-2017-13287. https://nvd.nist.gov/vuln/detail/CVE-2017-13287. Accessed: April 2026.

[18] NVD - CVE-2017-13288. https://nvd.nist.gov/vuln/detail/CVE-2017-13288. Accessed: April 2026.

[19] NVD - CVE-2017-13289. https://nvd.nist.gov/vuln/detail/CVE-2017-13289. Accessed: April 2026.

[20] CVE - CVE-2017-13315. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13315. Accessed: April 2026.

[21] CVE - CVE-2018-9431. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9431. Accessed: April 2026.

[22] CVE - CVE-2018-9471. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9471. Accessed: April 2026.

[23] CVE - CVE-2018-9474. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-9474. Accessed: April 2026.

[24] NVD - CVE-2021-0928. https://nvd.nist.gov/vuln/detail/CVE-2021-0928. Accessed: April 2026.

[25] NVD - CVE-2022-20135. https://nvd.nist.gov/vuln/detail/CVE-2022-20135. Accessed: April 2026.

[26] NVD - CVE-2023-20963. https://nvd.nist.gov/vuln/detail/CVE-2023-20963. Accessed: April 2026.

[27] Android Security Bulletin-February 2025. https://source.android.com/docs/security/bulletin/2025-02-01. Accessed: April 2026.

[28] NVD - CVE-2025-31175. https://nvd.nist.gov/vuln/detail/CVE-2025-31175. Accessed: April 2026.

[29] Zeinab El-Rewini and Yousra Aafer. Dissecting residual apis in custom android roms. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1598–1611. ACM, 2021.

[30] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2379–2396. USENIX Association, 2020.

[31] extract_android_ota_payload. https://github.com/cyxx/extract_android_ota_payload. Accessed: April 2026.

[32] Huan Feng and Kang G. Shin. Understanding and defending the binder attack surface in android. In Stephen Schwab, William K. Robertson, and Davide Balzarotti, editors, *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, pages 398–409. ACM, 2016.

[33] Firmware File - Database of Stock ROM (Flash File). https://firmwarefile.com/. Accessed: April 2026.

[34] AccountManagerService.java - Android Code Search. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/services/core/java/com/android/server/accounts/AccountManagerService.java;l=4942?q=AccountManagerService&ss=android%2Fplatform%2Fsuperproject. Accessed: April 2026.

[35] Google for Developers - from AI and Cloud to Mobile and Web. https://developers.google.com/. Accessed: April 2026.

[36] Google Play Store number of apps 2026. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Accessed: April 2026.

[37] Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. Preventing serialization vulnerabilities through transient field detection. In Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song, editors, *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, pages 1598–1606. ACM, 2021.

[38] HARMONYOS NEXT COMES WITHOUT A SINGLE LINE OF ANDROID CODE. https://gizchina.net/en/2024/01/19/harmonyos-next-predstavleno-bez-zhodnoho-ryadka-kodu-android/. Accessed: April 2026.

[39] Behnaz Hassanshahi and Roland H. C. Yap. Android database attacks revisited. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 625–639. ACM, 2017.

[40] Roee Hay. fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations. In William Enck and Collin Mulliner, editors, *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. USENIX Association, 2017.

[41] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 118–128. ACM, 2015.

[42] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1236–1247. ACM, 2015.

[43] Intents and intent filters. https://developer.android.com/guide/components/intents-filters. Accessed: April 2026.

[44] jadx. https://github.com/skylot/jadx. Accessed: April 2026.

[45] Java Data Types. https://www.w3schools.com/java/java_data_types.asp. Accessed: April 2026.

[46] Java Pathfinder. https://github.com/javapathfinder. Accessed: April 2026.

[47] Yiming Jing, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi. Checking intent-based communication in android with intent space analysis. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 735–746. ACM, 2016.

[48] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. FANS: fuzzing android native system services via automated interface analysis. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 307–323. USENIX Association, 2020.

[49] lpunpack. https://github.com/unix3dgforce/lpunpack. Accessed: April 2026.

[50] SmaliEx. https://github.com/testwhat/SmaliEx. Accessed: April 2026.

[51] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 543–558. USENIX Association, 2013.

[52] Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: April 2026.

[53] pacextractor. https://github.com/divinebird/pacextractor. Accessed: April 2026.

[54] Parcelable. https://developer.android.com/reference/android/os/Parcelable. Accessed: April 2026.

[55] Parcel. https://developer.android.com/reference/android/os/Parcel. Accessed: April 2026.

[56] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In Aurélien Francillon and Thomas Ptacek, editors, *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. USENIX Association, 2015.

[57] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android OEM compliance and customization. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 87–102. IEEE, 2021.

[58] CVE-2023-20963 Exploited by Chinese E-commerce App Pinduoduo. https://sensorstechforum.com/cve-2023-20963-android-pinduoduo-app/. Accessed: April 2026.

[59] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Trans. Softw. Eng. Methodol.*, 32(1):25:1–25:45, 2023.

[60] sdat2img. https://github.com/xpirt/sdat2img. Accessed: April 2026.

[61] android-simg2img. https://github.com/anestisb/android-simg2img. Accessed: April 2026.

[62] Prashast Srivastava, Flavio Toffalini, Kostyantyn Vorobyov, François Gauthier, Antonio Bianchi, and Mathias Payer. Crystallizer: A hybrid path analysis framework to aid in uncovering deserialization vulnerabilities. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, pages 1586–1597. ACM, 2023.

[63] After Huawei and Xiaomi, Vivo announced BlueOS its own mobile operating system. https://www.huaweicentral.com/after-huawei-and-xiaomi-vivo-announced-blueos-its-own-mobile-operating-system/. Accessed: April 2026.

[64] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards understanding android system vulnerabilities: Techniques and insights. In Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang, editors, *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 295–306. ACM, 2019.

[65] JingZheng Wu, Shen Liu, Shouling Ji, Mutian Yang, Tianyue Luo, Yanjun Wu, and Yongji Wang. Exception beyond exception: Crashing android system by trapping in "uncaught exception". In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 283–292. IEEE Computer Society, 2017.

[66] Xiaomi Developing Own Operating System Compatible with Android, Aims to Compete with Huawei's HarmonyOS and Eventually Google. https://www.gizmochina.com/2023/08/23/xiaomi-developing-smartphone-operating-system/. Accessed: April 2026.

[67] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Hai-Xin Duan. Intentfuzzer: detecting capability leaks of android applications. In Shiho Moriai, Trent Jaeger, and Kouichi Sakurai, editors, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 531–536. ACM, 2014.

[68] Yuqing Yang, Mohamed Elsabagh, Chaoshun Zuo, Ryan Johnson, Angelos Stavrou, and Zhiqiang Lin. Detecting and measuring misconfigured manifests in android apps. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 3063–3077. ACM, 2022.

[69] Yunfan Zhan, Qidan He, Yijun Wang, and Xiuzhen Chen. PMDET: automated detection tool of android parcel mismatch. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 250–254. IEEE, 2024.

[70] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. Exploit the last straw that breaks android systems. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2230–2247. IEEE, 2022.

A App Analysis Results

Out of the 100 apps from the Tools category, we identify 46 apps that are vulnerable to the malformed Parcel DoS attack. Table 7 lists their package names with the number of their downloads from the Google Play Store. We manually verify each of the 46 apps and confirm that they all can be crashed on a Google Pixel 6a device with the latest Android system installed.

Table 7: Vulnerable apps to the malformed Parcel DoS attack identified by our analysis in the Tools category on Google Play Store.

App Package Name	Downloads
com.google.android.inputmethod.latin	5B+
com.google.android.apps.translate	1B+
com.google.android.dialer	1B+
com.google.ar.lens	1B+
com.lenovo.anyshare.gps	1B+
com.sec.android.easyMover	500M+
com.transsion.phonemaster	500M+
com.psiphon3.subscription	100M+
com.avast.android.mobilesecurity	100M+
com.kms.free	100M+
com.lookout	100M+
com.antivirus	100M+
com.google.android.apps.adm	100M+
com.google.android.apps.authenticator2	100M+
com.google.android.apps.kids.familylink	100M+
com.mate.vpn	100M+
com.piriform.ccleaner	100M+
com.tplink.tether	100M+
com.vzw.hss.myverizon	100M+
ru.zdevs.zarchiver	100M+
free.vpn.unblock.proxy.vpn.master.pro	50M+
com.avast.android.cleaner	50M+
com.duckduckgo.mobile.android	50M+
com.expressvpn.vpn	50M+
com.free.turbo.unlimited.touch.vpn	50M+
com.free.vpn.super.hotspot.open	50M+
com.google.android.apps.paidtasks	50M+
com.symantec.mobilesecurity	50M+
com.avast.android.vpn	10M+
com.avira.android	10M+
com.eset.ems2.gp	10M+
com.s.antivirus	10M+
com.security.xvpn.z35kb	10M+
com.windscribe.vpn	10M+
com.bitdefender.security	10M+
com.bloodtracker.smartbp	10M+
com.symantec.securewifi	10M+
com.atlasvpn.free.android.proxy.secure	5M+
com.avg.android.vpn	5M+
com.kaspersky.secure.connection	5M+
com.privateinternetaccess.android	5M+
com.microsoft.scmx	1M+
io.privado.android	1M+
com.beepassvpn.free.vpn.secure	1M+
com.avast.android.antivirus.one	500K+
com.eset.etvs.gp	100K+

B Firmware Dataset

We collected 324 firmware samples from various sources. Table 8 summarizes the list of the vendor names and their download websites.

C Parcelable Class in HarmonyOS

We find that a similar Parcel mechanism exists in HarmonyOS NEXT, which is an emerging mobile operating system by Huawei. We successfully implement a Parcelable class, as shown in Figure 11, in our testing HarmonyOS app. Similar to Android, the Parcelable class contains two functions, `marshalling()` and `unmarshalling()`, which are equivalent to `writeToParcel()` and `readFromParcel()` in Android. We implement the class asymmetrically to read an int and write a long and trigger a data mismatch within our test app.

Table 8: The websites and corresponding vendors from which we downloaded the Android firmware samples.

Vendor	URL
Google	https://developers.google.com/android/images
Samsung	https://www.sammobile.com/
Xiaomi	https://xiaomifirmwareupdater.com/miui/
OnePlus	https://yun.daxiaamu.com/OnePlus_Roms/
Vivo	https://vivofirmware.com
Motorola	https://motostockrom.com
Lenovo	https://firmwarefile.com/
Sony	https://xperiastockrom.com/
ZTE	https://ztefirmware.com/

```
class MyParcelable implements rpc.Parcelable {
  int1: number = 0;
  int2: number = 0;
  constructor(int1: number, int2: number) {
    this.int1 = int1;
    this.int2 = int2;
  }
  marshalling(messageSequence: rpc.MessageSequence): boolean {
    messageSequence.writeLong(this.int1);
    messageSequence.writeInt(this.int2);
    return true;
  }
  unmarshalling(messageSequence: rpc.MessageSequence): boolean {
    this.int1 = messageSequence.readInt();
    this.int2 = messageSequence.readInt();
    return true;
  }}
```

Figure 11: Example of a Parcelable class in HarmonyOS.

D Detailed Firmware Unpacking

Unpacking firmware images. The Android ecosystem contains many vendor-specific firmware packaging methods, and vendors' methods have also changed over time. For this reason, unpacking large numbers of Android firmware is a very involved effort, requiring knowledge of the many formats that vendors use and of tools that can be used to unpack, and in some cases decrypt, various file formats found in firmware. Commonly, the Android Framework files can be found in the system image "system.img", which is an ext4, or ext2 in older firmware, filesystem image found in the vendor's packaged firmware. We consider firmware to have been unpacked once we are able to successfully mount the system image. In many cases, the system image will be packaged as a partition within a super image file "super.img", which can then be extracted using lpunpack [49]. In some cases, the system image file will be compressed as an Android Sparse Image, in which case an open-source tool, simg2img [61], can be used to unpack the filesystem image from the sparse image format to a raw image that can be mounted. Firmware from vendors such as Xiaomi, Vivo, and OnePlus sometimes split the system image into many files with a ".dat" extension and provide a "transfer list" file that contains information on how to combine the files to construct the system.img; the sdat2img [60] tool can be used to reconstruct these split filesystem images. Some firmware from vendors such as Xiaomi and Vivo will be packed into an OTA file named payload.bin that can be unpacked using extract_android_ota_payload [31]. Sony uses the SIN file format to pack their firmware, which can be extracted using the anyxperia_dumper tool [3]. Vendors such as Lenovo will sometimes pack their firmware inside a "pac" file, which can be unpacked using the pacextractor tool [53].

Extracting Android Framework. Once the firmware is unpacked, the extraction of the Android Framework files is universal across vendors, but not across Android versions. For recent Android versions, Android 10 through 14, this process is simple: the Framework code is found in the "system/framework/framework.jar" file. For older Android versions, Android 7 through 9, the Framework code can be found in "system/framework/arm/arm64/boot-framework.oat" from which we extract smali files using baksmali [5], then use jadx [44], a Java decompiler, to convert the smali to decompiled Java source. For the oldest Android version we processed, Android 6, the Framework code is found in "system/framework/arm/arm64/boot.oat" which we convert to DEX files using the oat2dex tool [50].

E Detailed Parcelable Class Analysis

AST generation and pruning. To extract information from the reading and writing methods in a Parcelable class, we initiate the process by applying a Java AST generator to the source code of the Parcelable class. Subsequently, we further prune the generated AST in order to remove content that is not related to the reading and writing methods, such as constructors and helper functions. The pruned AST only contains the reading and writing methods.

Type inconsistency identification. Utilizing the pruned AST information, we determine whether the reading and writing methods incorporate if-else control flows. If absent, we proceed with basic mismatch identification and produce the results. However, if the reading and writing methods include if-else control flows, we conduct non-trivial mismatch checking following the completion of the basic mismatch identification.

To identify the basic mismatch, we first extract the data types in the reading and writing statements as well as their sequences for further analysis. Using the vulnerable case in Figure 2 as an example, we extract `String`, `Int`, `Int`, and `ByteArray` as the data types and sequence from the reading method, and extract `String`, `Long`, `Int`, and `ByteArray` as the data types and sequence from the writing method. We then compare whether the data types and sequences from the reading and writing methods are consistent and report the consistency in the results. In this example in Figure 2, they are inconsistent, and we report the alert in the results.

Since Android allows the usage of diverse data types in Parcel objects, different data types in reading and writing statements may have the same utility and functionality. For instance, `writeList()` is considered to be consistent with `readList()`, but it is also considered to be consistent with `readArrayList()`. Simply comparing the consistency of data types is not sufficient to handle all cases. To solve this challenge, we create a reference database and incorporate all allowable use cases of different data types permitted by Android. We reference the database when performing the basic mismatch identification, which accurately identifies truly harmful cases of inconsistency.

Value constraint inconsistency identification. Using the vulnerable case in Figure 4a as an example, non-trivial mismatches

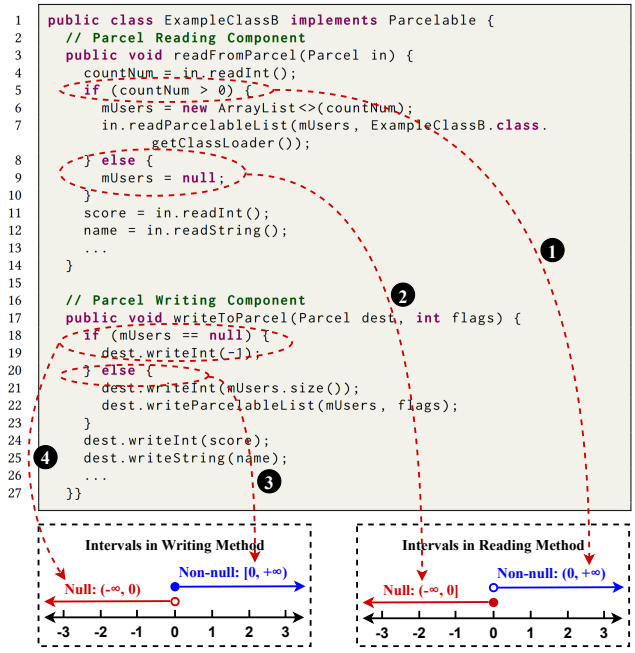


Figure 12: Interval mapping. ❶ and ❷ semantically convert the condition of the if-else control flow in the reading method to intervals. ❸ and ❹ semantically convert the condition of the if-else control flow in the writing method to intervals. Our analysis also supports if-else-if-else control flows.

due to imperfect if-else control flows are challenging to identify for several reasons. First, there are no direct data type inconsistencies to identify. Second, the if-else control flows in the reading and writing methods are not directly correlated and do not interact with each other. For instance, line 5 in Figure 4a uses `countNum` as the condition checking in the control flow of the reading method, while line 18 uses `mUsers` as the condition checking in the control flow of the writing method. However, `countNum` and `mUsers` are different variables and they are not directly correlated, although `countNum` separately represents the length of `mUsers` in the context. Third, while it is possible to enumerate all possible cases until finding the corner case that leads to the mismatch, this approach is neither efficient nor scalable.

To address this challenge, we introduce an interval mapping analysis. We utilize the vulnerable case in Figure 4a as an example and demonstrate the interval mapping analysis in Figure 12. The interval mapping analysis extracts information from the conditions of the if-else control flows in the reading and writing methods based on the pruned AST information. In the meantime, it considers the semantic context and maps the intervals for non-null conditions and null conditions. We now describe the analysis in four steps as shown in Figure 12.

Reading method analysis (❶ and ❷). To analyze the reading method, we extract the information of the if-else control flows from the reading method, as demonstrated in ❶ and ❷ in Figure 12. We first identify that the condition for entering the if block is that the value of a variable is greater than zero. By considering the semantic context, which is the statement to set `mUsers` to null in the else block, we identify that the aforementioned condition greater than zero corresponds to non-null, while the condition less than or equal to zero corresponds to null. We then further infer that the interval corresponding to non-null in the reading method is zero exclusive to positive infinity, and the interval corresponding to null in the reading method is negative infinity to zero inclusive as shown in Figure 12.

Writing method analysis (❸ and ❹). Subsequently, to analyze the writing method, we extract the information of the if-else control flows from the writing method, as demonstrated in ❸ and ❹ in Figure 12. We first identify that the condition for entering the if block is that the variable `mUsers` is null. By considering the semantic context, which is the statement to write an int value -1 in the if block and writing statements in the else block, we infer that the condition less than zero corresponds to null, while the condition greater than or equal to zero corresponds to non-null. As illustrated in Figure 12, we then further infer that the interval corresponding to non-null in the writing method is zero inclusive to positive infinity, and the interval corresponding to null in the writing method is negative infinity to zero exclusive.

We also generalize our analysis to support if-else-if-else control flow analysis. Subsequently, we perform a comparison of the intervals of the reading and writing methods. In this example in Figure 12, the intervals are inconsistent because the boundary value zero falls into different intervals. Specifically, the boundary value zero falls into the null interval in the reading method, while it falls into the non-null interval in the writing method.

As we mentioned in Section 4.2, changing line 5 in Figure 12 to `if (countNum >= 0) {` fixes the vulnerability. We validate the interval mapping analysis on the fixed case. Since the condition in line 5 is changed to greater than or equal to zero, the boundary value zero now falls into the non-null interval in the reading method. The intervals in the reading and writing methods are consistent, which verifies the interval mapping analysis. After checking the non-trivial mismatch, we incorporate the consistency report into the results. If the intervals in the reading and writing methods are inconsistent, we report the alert in the results.