Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2020, 7(5):407-413**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Detecting App Data Clearance and Uninstallations in Android: Approaches for Fraud Detection

Sambu Patach Arrojula

Email: sambunikhila@gmail.com

Abstract: Fraud detection in Android applications is a crucial component for many industries, particularly those involved in finance, security, and secure communications. A user's ability to clear app data or uninstall and reinstall the app introduces vulnerabilities that fraudsters can exploit to avoid detection. Unfortunately, Android does not provide direct notifications for data resets or uninstallation events, posing a challenge for developers who need to track these actions to update fraud detection algorithms. This paper discusses various approaches that can be used to infer data clearance and app uninstallation events in an Android environment. By analyzing techniques such as tracking flags in shared preferences, leveraging backend storage, and employing companion apps, this article evaluates each method's effectiveness and limitations. We propose a multi-layered approach that combines backend integration with companion app functionality to create a more robust fraud detection mechanism, balancing the trade-offs between security, performance, and user experience.

Keywords: Android application data reset event, Android application uninstall event, fraud detection in android app, manage space activity, companion application.

1. Introduction

In Android, users can easily reset app data or uninstall an app, which removes all internal data, preferences, and configurations. For typical apps, this action might not pose any significant problems, but for secure or fraud-sensitive applications, the implications are critical. Clearing data or reinstalling the app could allow malicious users to reset fraud detection parameters, making it easier for them to bypass protections or commit fraud undetected. Fraud detection algorithms need timely updates about such events to mitigate risks and take corrective actions.

However, Android does not provide a straightforward API or event listener to detect when app data has been cleared or the app has been uninstalled and reinstalled. This lack of direct access makes it necessary for developers to use creative workarounds. In fraud-sensitive environments, especially those involving financial transactions, user verification, or sensitive communications, the ability to detect and respond to app resets or uninstallations becomes a critical aspect of maintaining security. This article explores methods for detecting these events and discusses the advantages and drawbacks of each approach.

2. Approach

Detecting app data clearance and uninstallations is inherently challenging due to the way Android handles these events. Unlike other event tracking, when an app is uninstalled or its data is cleared, the app process is terminated immediately, and in the case of uninstallation, the app's package is completely removed from the device. Moreover, Android does not provide a straightforward mechanism for apps to preserve information

across uninstallations or data resets that can be accessed upon reinstallation or relaunch. This lack of direct support means that developers cannot rely on simple callbacks or system notifications to detect these events.

Given these challenges, developers in the industry have adopted various approaches to infer when a data reset or uninstallation has occurred. However, each of these methods has limitations, and none offer a complete solution on their own.

Here we will explore a few approaches and evaluate them based on three key aspects:

1. Reliability: How likely is this approach to accurately detect the event? Are there any corner cases where it may fail?

2. Timeliness: How quickly can the approach detect the data reset or uninstallation?

3. Balance between practicality and reliability: How feasible is the approach in a real-world scenario, considering factors like system resources, user permissions, and ease of implementation?

Recognizing this gap, we explore a combination of existing techniques to develop a hybrid method that enhances the ability to detect these events more reliably.

Our proposed approach aims to:

1. Identify the occurrence of data resets and uninstallations as promptly as possible to minimize the threat surface and allow fraud detection algorithms to take immediate action.

2. Utilize a combination of client-side and server-side mechanisms, including backend integration and a companion app, to track and infer these events effectively.

3. Balance the need for security with practical considerations, such as minimizing performance overhead, preserving user experience, and adhering to permission restrictions.

By integrating multiple strategies, we aim to provide a more robust solution that overcomes the individual limitations of each method. This hybrid approach can better support fraud detection algorithms in critical security applications by providing timely and accurate information about app data clearance and uninstallations.

3. Evaluations

1. Tracking Through Shared Preferences or Local Files

- **Approach:** The app stores a flag (or file) in its local storage, indicating that the app has previously been launched. Upon each app launch, the app checks whether this flag exists. If it does not, this could indicate that the app data has been cleared.
- **Insights:** While this is a lightweight, simple approach that does not require any external dependencies or additional permissions. This method cannot distinguish between a first-time installation and a data reset, as both scenarios result in the absence of the flag. Additionally, it offers no insight into uninstallation events.
- **Usecases:** As this approach will fail to distinguish data resets from first installations. This could be a candidate for non fraud critical applications.

2. Frequent Silent Push Notifications Through FCM

- **Approach:** • The backend system integrates with Firebase Cloud Messaging (FCM) and schedules frequent silent push notifications. These silent messages don't trigger any user-visible actions but serve as a "heartbeat" to confirm that the app instance is still active. If the app is uninstalled or data is cleared, the FCM push will fail, indicating to the backend that the app is no longer active.
- **Insights:** While this approach doesn't require the app to be launched again to derive such info at backend, it cannot distinguish between app data resets and app uninstallation because both events result in the failure of the push notification. Additionally, detection is not immediate i.e. delayed because it relies on scheduled push notifications.
- **Usecases:** As this approach will fail to distinguish data resets from uninstallation. This could be a candidate for semi fraud critical applications.

3. External Flagging with Backend Storage

- **Approach:** To distinguish between fresh installations and data resets, the app stores a flag on a backend server associated with a unique device identifier. Upon each launch, the app checks for the presence of this flag on the server.
- **Insights:** While this is approach capable of distinguishing between first-time installation and a data reset it can not identify if the data missing is because of uninstall or just clearing data. Also is not so light approach to

be implement i.e. we need whole backend system to be designed and integrated with client and client has to finish its primary setup with backend if needed(like token exchange etc).

- **Usecases:** As this approach will fail to identify uninstallations this could be a candidate for moderate fraud critical applications.

4. Manage Space Activity

- **Approach:** Implement a Manage Space Activity that is triggered when the user attempts to clear the app's data through the system settings. This activity allows the app to perform actions before the data is cleared, such as notifying the backend server.
- **Insights:** While this approach can detect a data reset immediately it cannot identify app uninstallations.
- **Usecases:** As this approach will fail to identify uninstallations this could be a candidate when immediate detection of data resets is critical.

Implementation details:

First, you need to declare the ManageSpaceActivity in your app's AndroidManifest.xml. You also need to associate this activity with the android.intent.action.MANAGE_PACKAGE_STORAGE intent so that the system knows this activity is responsible for managing the app's storage.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.storageapp">

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">

    <!-- Other activities -->

    <!-- Manage Space Activity -->
    <activity android:name=".ManageSpaceActivity">
      <intent-filter>
        <action
android:name="android.intent.action.MANAGE_PACKAGE_STORAGE" />
        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
    </activity>

  </application>
</manifest>
```

Next, create the ManageSpaceActivity in your app. This activity should present the user with options to clear cached data or other unnecessary files stored by the app.

Here's an example of a simple ManageSpaceActivity:

```
public class ManageSpaceActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Show a dialog to the user with options to clear data
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle("Manage Storage");
    builder.setMessage("You can clear app's cached data or other stored files.");

    // Option to clear cached data
    builder.setPositiveButton("Clear Cache", new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        clearAppCache(); // Logic to clear Cache
      }
    });

    // Option to clear stored files
    builder.setNegativeButton("Clear All Data", new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        updateDataResetEvent() // Here we update fraud algo about data reset
        clearAppData(); // logic to clear Complete data
      }
    });

    // Option to cancel
    builder.setNeutralButton("Cancel", new DialogInterface.OnClickListener() {
      @Override
      public void onClick(DialogInterface dialog, int which) {
        dialog.dismiss();
        finish(); // Close the activity
      }
    });

    builder.show();
  }
}
```

5. Companion App for Uninstallation Detection

- **Approach:** Introduce a lightweight companion app installed alongside the primary app. Both apps monitor each other's installation status by listening for package removal broadcasts. If the primary app is uninstalled, the companion app detects this event and notifies the backend server.
- **Insights:** While this approach can detect a uninstallations immediately it brings additional dependency of companion app into the system.
- **Usecases:** As this approach will fail to identify data rest and brings additional dependency this could be a candidate when immediate detection of uninstallation is critical.

Developer can opt any approach to install its companion (we recommend bundling it within asset) with proper user consent. Following are BroadcastReceiver setup of both primary and companion apps.

```
package com.example.strorageapp

import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.widget.Toast

class UninstallReceiver : BroadcastReceiver() {
  override fun onReceive(context: Context, intent: Intent) {
    val packageName = intent.data?.schemeSpecificPart
    if (packageName == "com.example.storageapp.companion") {
      // Logic to justify user about the companion app and request to re-
install
      // if he still rejects our system will
      // runs without the availability of companion app
    }
  }
}
```

```
package com.example.strorageapp.companionapp

import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.widget.Toast

class UninstallReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val packageName = intent.data?.schemeSpecificPart
        if (packageName == "com.example.storageapp ") {
            // Logic to update backend about uninstallation event
        }
    }
}
```

6. Combining Approaches in a Hybrid Method

As every method has its own pros and cons we are proposing a hybrid method through which the system has **more coverage of reliable fraud events in relatively more use cases.**

Proposal: We suggest a hybrid approach we combine these three approaches NoData flag at backend + Manage Space Activity + Companion app to get a mixed result from multiple aspects to improve detection accuracy for both data resets and uninstallations.

1. Implementation:

O On App Launch: The app checks for a local "no-data" flag in its internal storage. If the flag is missing, it indicates a possible data reset or fresh installation.

O Backend Verification: The app sends a "no-data" flag along with a unique device ID to the backend server. The server checks its database for records associated with the device ID.

O First-Time Installation: If no record exists, the server registers this as a first-time installation. If otherwise it would be a data missing case but system not sure if its because of data reset or uninstallation.

O Data Reset Detection: If a "data reset" flag was previously set via the Manage Space Activity, the server confirms a data reset event and clears the flag.

O Uninstallation Detection: If an "uninstallation" flag was received from the companion app, the server recognizes an uninstallation event and clears the flag.

O Unexpected Cases: If none of the above flags are set, i.e. neither ManageSpaceActivity nor CompanionApp flag, the server may infer that the data reset occurred via non-standard methods (e.g., ADB commands), which could be significant for fraud detection.

2. Advantages:

O Comprehensive Detection: By combining local checks, backend verification, and companion app monitoring, this approach enhances the detection of both data resets and uninstallations.

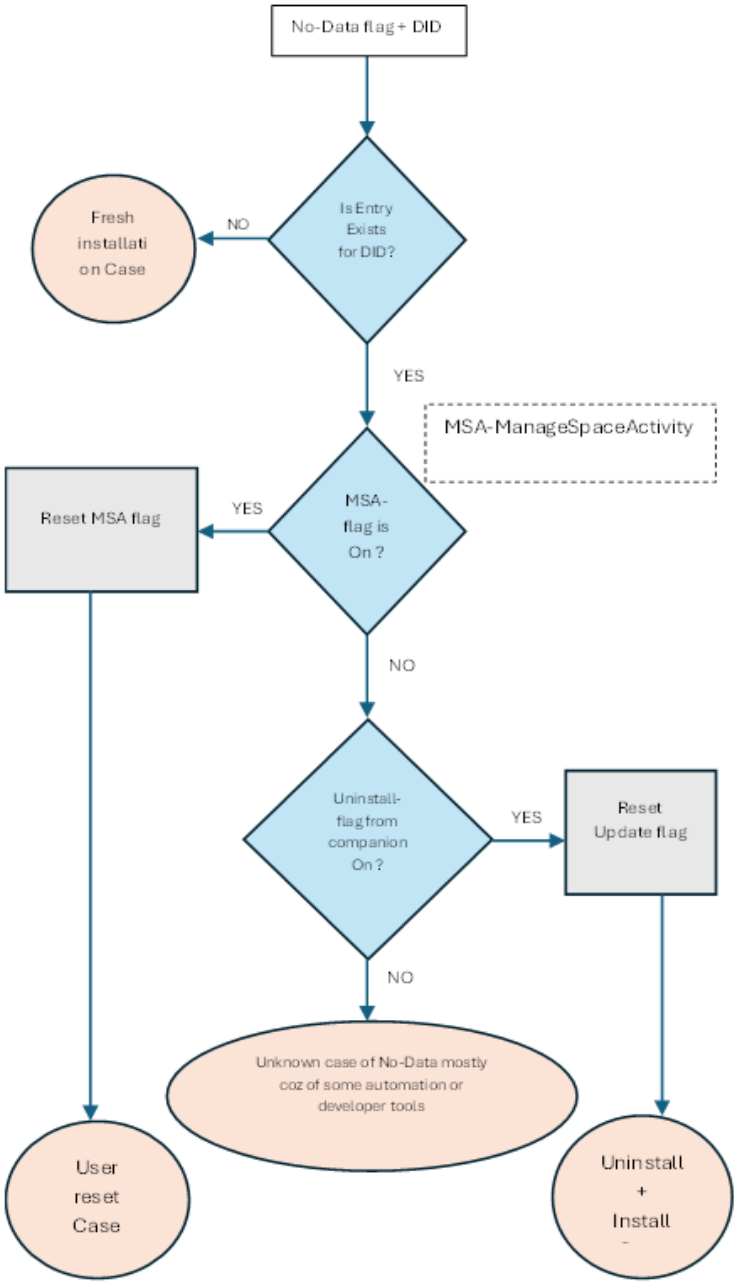
O Timely Updates: The use of Manage Space Activity and companion app notifications ensures that the backend is informed promptly.

3. Disadvantages:

O Complex Implementation: Integrating multiple methods increases development complexity.

O User Acceptance: Users may be hesitant to install a companion app or grant necessary permissions.

4. Use Case: Best suited for critical applications where security is paramount, and users are likely to accept additional measures for enhanced protection.



4. Conclusions

Detecting app data clearance and uninstallations in Android is a complex challenge due to the lack of direct support from the operating system. Individual methods, such as tracking local flags or using backend storage, have limitations in distinguishing between first-time installations, data resets, and uninstallations. By adopting a hybrid approach that combines multiple strategies—local flagging, backend verification, Manage Space Activity, and a companion app—we can achieve a more robust detection mechanism.

This multi-layered approach allows fraud detection algorithms to receive timely and accurate information about app data clearance and uninstallations, enabling them to mitigate risks more effectively. While this method increases implementation complexity and may impact user experience, it provides a comprehensive solution for critical security applications where the benefits outweigh the drawbacks. Developers must balance security needs with practical considerations, tailoring the approach to their app's specific requirements and user base.

By enhancing the ability to detect these events promptly, applications can significantly reduce the threat surface and improve the overall effectiveness of their fraud detection systems. This approach supports the ongoing

efforts to secure Android applications against increasingly sophisticated fraud tactics, contributing to safer and more trustworthy app ecosystems.

References

[1]. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-liu_bin.pdf
[2]. https://dl.acm.org/doi/abs/10.1145/2594368.2594391
[3]. https://www.techaheadcorp.com/blog/data-analytics-fraud-prevention/
[4]. https://clevertap.com/blog/track-app-uninstalls-effectively/
[5]. https://www.ndss-symposium.org/wp-content/uploads/2017/09/life-after-app-installation-data-still-alive-data-residue-attacks-android.pdf
[6]. https://medium.com/@_foso_/what-is-a-managespaceactivity-on-android-26530ba4117b