

A Security Analysis of the Multi-User Ecosystem in Android Framework

by

Muhammad Shahpar Nafees Khan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Muhammad Shahpar Nafees Khan 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Android framework’s multi-user ecosystem introduces significant security challenges, particularly in the enforcement of user-specific access control checks. While previous research has highlighted flaws in Android’s access control mechanism, these efforts often overlook the complexities introduced by vendor customization and the unique demands of a multi-user environment. In this thesis, we conduct a systematic analysis of the Android Open Source Project (AOSP), identifying key patterns regulating multi-user access control implementations. We use these patterns to develop MVP, a static analysis tool that examines vendor ROMs for missing user-specific access control checks in custom ROMs. For example, our analysis reveals that Android’s multi-user environment is susceptible to cross-user attacks; sensitive data can be shared between profiles, and non-privileged users can manipulate privileged system settings. These findings underscore the need for rigorous enforcement of access control mechanisms to mitigate security risks in Android’s multi-user environment.

Acknowledgements

This work would not have been possible without the unwavering support of my family and friends, who stood by me throughout this journey.

I am deeply grateful for the support, guidance, and mentorship of my supervisor, Prof. Yousra Aafer, throughout my degree. I would also like to thank Prof. Diogo Barradas and Prof. Chengnian Sun for being part of my Master's thesis committee and for providing invaluable feedback.

I am thankful to my parents, Sadia and Iqbal, and my brother, Ammar, for always allowing me to carve my own path and supporting me through every step of my growth. A special thank you to Parjanya for his guidance, without which I would not have learned as much as I did. I am also grateful to all of my friends, especially Mashal, Monum, Adnan, Hasaan, and Sharjeel, for their constant encouragement and for keeping my morale high throughout this process. Finally, I want to thank Haleema for believing in me and always standing by my side.

Dedication

This is dedicated to my family, friends, and my loved one.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	ix
List of Tables	x
List of Code Listings	xi
1 Introduction	1
2 Background	3
2.1 The Implementation of the Multi-User Ecosystem	3
2.2 Access Control Enforcement in the Multi-User Framework	4
2.3 Security Hazards under OEM Customization	5
3 Motivation	6
3.1 The Impact of Multi-User Android Ecosystems in Daily Life	6

3.2	Problem: Why Are Vendors Failing to Protect APIs from Cross-User Attacks?	6
3.3	Motivation Example	7
4	Characterizing User Specific Access Control Checks	10
4.1	Scope of Manual Analysis	10
4.2	Multi-User Access Control Patterns	10
4.2.1	Pattern 1: User Status Management	11
4.2.2	Pattern 2: User Ownership Enforcement	14
4.2.3	Pattern 3: User Privilege Separation	16
5	Design and Implementation	19
5.1	Collection Phase	20
5.2	Analysis Phase	24
5.2.1	Extracting Access Control Checks	24
5.2.2	Detecting Pattern 1	24
5.2.3	Detecting Pattern 2	24
5.2.4	Detecting Pattern 3	25
6	Evaluation	26
6.1	Analysis Complexity and Configuration	26
6.2	Dataset Collection	27
6.3	RQ1: How prevalent are user-specific access control checks in AOSP and vendor ROMs?	28
6.4	RQ2: To what extent do vendors increase the cross-user attack surface? . .	30
6.5	RQ3: How do cross-user vulnerabilities evolve over time?	32
6.6	Case Study 1: User Data Leak Across User Profiles	33
6.7	Case Study 2: Privileged User Settings Updated by Non-privileged Users .	34

7	Related Works	35
7.1	Android Framework Multi-user	35
7.2	Inconsistent Access Control	36
7.3	Permission Maps	36
7.4	OEM Customization	37
8	Limitations	38
8.1	Vendors Enforcing Diverse Custom Access Control Checks	38
8.2	Missing Ground Truth	38
9	Conclusion	40
	References	41

List of Figures

5.1	End-to-end design of MVP	19
6.1	Number of vulnerable APIs missing user-related access control checks across different vendors and Android versions	32

List of Tables

5.1	Access Control checks extracted by the Analyzer Module of MVP	24
6.1	Average Counts of APIs and User-specific Access Control Checks per ROM Across Vendors	28
6.2	Access Control Pattern Counts by Vendor for Android Versions 10 to 13 .	30
6.3	Corpus of System Service APIs missing user-specific access control checks, classified through manual testing into two categories: Safe and Triggerable	31

List of Code Listings

3.1	AOSP enforcing proper cross user access control checks	8
3.2	Samsung API leaking secure folder policy data across users	8
3.3	Vivo API enabling privileged setting changes by an under-privileged user .	8
4.1	AOSP APIs showcasing user status management patterns	11
4.2	System services dynamically updating global userId fields through Broadcast Receiver	13
4.3	System services dynamically updating global userId fields through Lifecycle Method	13
4.4	AOSP API showcasing a user ownership check	15
4.5	(A) AOSP API showcasing user privilege check	17
4.6	(B) AOSP API showcasing user privilege check	17
5.1	(A) User-related method that facilitates the identification of userId fields and parameters	22
5.2	(B) User-related method that facilitates the identification of userId fields and parameters	22
6.1	getLastUserActivityTime() in PowerManagerService	33
6.2	saveDarkModeStatusForMultiLand() in VivoNightModeService	34

Chapter 1

Introduction

Operating systems include a multi-user ecosystem to allow shared usage of a device across different physical users. Although Android is based on Linux, which inherently possesses multi-user capabilities, Android first introduced its multi-user capabilities in 2012 (version 4.2) [48]. While multi-user support allows users to have separate accounts and resources, it also creates more choke points in the Android framework that can lead to cross-user attacks.

Multi-user systems are known to be susceptible to numerous attack vectors, including, but not limited to, privilege escalation and cross-user data leakage [44]. Like other operating systems, Android also suffers from the proliferation of these attack vectors due to a lack of standardization of cross-user access control mechanisms, particularly, in light of the under-regulated extensive OEM customization [29].

Researchers have undertaken significant efforts to evaluate Android access control implementation. Some approaches resort to static inconsistency analysis [18], dynamic analysis [27], or probabilistic inference [30]. However, none of these efforts is specifically tailored to the multi-user ecosystem. The latter represents a new paradigm and necessitates a more targeted approach to unveil underlying access control anomalies.

Ratazzi et al. pioneered the systematic study of the Android multi-user ecosystem, mapping its mechanisms and identifying vulnerabilities through hypothesis testing and manual analysis [48]. Gorski's et al. further explored the ecosystem through semi-automated approaches to identify permission re-delegation vulnerabilities, leading to the discovery of various cross-user attacks [38, 36, 39, 37]. While these previous efforts are commendable, they have inherent limitations. Ratazzi's work, although systematic, lacked automation and tackles very old versions of Android where the multi-user implementation was still

in its earlier stages. Gorski’s more recent work is automated but does not directly focus on multi-user vulnerabilities, leaving the evaluation of such vulnerabilities incomplete and limited to those that can be detected through general inconsistency analysis. In this study, we conduct a systematic evaluation and vulnerability analysis of the multi-user ecosystem within the Android framework. We gather the key framework artifacts that manage user-specific access control enforcement in the multi-user ecosystem, identify user-related access control patterns, and pinpoint potential violations of such patterns through static analysis.

We carried out a targeted manual analysis of AOSP codebases, examining more than 2,500 system service APIs across Android versions 13 and 14. Through our investigation, we identified user-related access control patterns implemented by AOSP to regulate resource access in the multi-user environment. To detect potential vulnerabilities, we developed a static analysis tool, MVP, that identifies violations of these patterns in custom OEM framework codebases. We analyzed 73 ROMs from four different vendors, uncovering 79 APIs missing user-specific access control checks. To demonstrate the impact of these missing checks, we selected APIs in ROMs for which we had a physical device and developed Proof-of-Concepts (PoCs) attacks. Two APIs resulted in the leakage of sensitive data across user profiles and the mutation of privileged settings by non-privileged users in a Samsung S20 FE 5G and a Vivo Y33s device respectively. These vulnerabilities were acknowledged by the respective vendors. Additionally, we conducted a longitudinal study to track the evolution of APIs missing user-related access control checks. On the positive side, the longitudinal analysis reveals some vendors demonstrated greater consistency in enforcing user-related access control checks in later versions compared to others.

The contributions of this work are threefold:

- We conduct an exploratory manual analysis of AOSP, synthesizing user-related access control into 3 comprehensive patterns that are integral to the Android multi-user ecosystem.
- We design and implement MVP, a static analysis framework, to encode the identified patterns and detect instances of their violations; where a vendor fails to adequately implement a given pattern.
- We apply MVP to analyze 73 Android ROMs, uncovering 79 instances of APIs lacking user-related access control checks.

Chapter 2

Background

This chapter outlines the details of the multi-user ecosystem in Android, focusing on access control mechanisms and their associated security risks.

2.1 The Implementation of the Multi-User Ecosystem

While Android is based on the Linux OS, which inherently supports multi-user functionality, the multi-user feature was not incorporated into the Android ecosystem until version 4.2 (2012) [48, 43]. The introduction of multi-user functionality enhances Android’s support for shared use, allowing multiple users to operate on the same device through different user profiles [10].

The Android framework, a key component of Android’s architecture, hosts system services that handle long-running system operations and provide APIs that the application layer can use to perform specific tasks [52]. One such service is the **UserManagerService** (UMS) responsible for managing users and user details. UMS assigns unique `userIds` to differentiate between users. The default `userId` for the system user (also the device owner) is 0, while the next added user is assigned a `userId` of 10, with subsequent `userIds` increasing by 1 for each additional user. Certain negative `userIds` implicitly refer to specific users: (1) -1 represents all users, (2) -2 represents the current user, and (3) -10000 represents the null user.

Android Unique User IDs (UIDs) are crucial for keeping applications isolated from each other and for managing security and resources. Since multiple users may share apps, the `appId` alone no longer functions as a unique identifier. To support app differentiation across

users, the UID is generated by combining the unique `userId` with the `appId`. The UID is calculated as follows:

$$\text{UID} = \text{userId} \times \text{PER_USER_RANGE} + (\text{appId} \bmod \text{PER_USER_RANGE})$$

To keep track of the currently running user, Android system services might declare a global field (hereafter referred to as “*global userId*”). The field is initialized at the system startup (specifically, during the initialization of a given system service) and updated upon user-related events, such as user switches [26].

2.2 Access Control Enforcement in the Multi-User Framework

Single-user access control mechanisms are inadequate for regulating multi-user resource access. Therefore, starting from version 4.2, Android implement multi-user access control to enforce the following three aspects:

1. User Context Separation: Android must ensure that data and resources belonging to different users are clearly isolated; e.g., a user should not access another user’s data unless explicitly allowed. To achieve this, the Android framework uses `userId` checks that check for the calling user through system methods, such as `UserHandle.getCallingUser()`, to maintain proper user context separation.

2. User Status Verification: In a multi-user system, it is imperative to restrict the functionalities of the background user affecting the foreground user; e.g., a background user should not access the foreground user’s resources. To achieve this, Android uses global `userId` to enforce current user checks and prevent unauthorized access of active users’ resources by the background user.

3. Privilege Separation: Privilege escalation occurs when an underprivileged user gains unauthorized access to a privileged resource, potentially compromising the security and integrity of the device and necessitating strict privilege separation [64, 67]. Privilege separation is achieved through user-related access control checks. Users are assigned unique `userId`s that dictate their privileges. A system user with `userId` 0 has higher privileges than secondary users (i.e., those with `userId`s ≥ 10). System permissions, such as `INTERACT_ACROSS_USERS`, can be further used to ensure that only user apps with special privileges are allowed to perform an operation across users.

2.3 Security Hazards under OEM Customization

Original Equipment Manufacturers (OEMs) acquire vanilla AOSP base images and then customize them according to their needs, for example, adding a custom drivers/APIs for accessing a new sensor) The customization spans the entire system, involving modifications to the existing code and/or the addition of new custom code [46].

Due to its highly underregulated nature, OEM customization has been reported to be detrimental to Android security [61]. Such security stand is likely exacerbated as code complexity increases under the multi-user ecosystem. Therefore, there is a need for a systematic/automated framework to evaluate multi-user specific customization. Such evaluation would ensure proper and secure implementation of multi-user access control enforcement in different OEMs, particularly during the introduction of new APIs.

Prior Work: In 2014, Ratazzi et al. analyzed security vulnerabilities in Android’s multi-user feature and proposed mitigation strategies. They highlighted issues related to improper access control, which could allow inactive users to spy on active users, stressing the need for robust security measures [48]. Additionally, numerous efforts have been made to detect vulnerabilities arising from inconsistent access control enforcement w.r.t. multi-user implementation [48, 18, 27, 38]. Despite the fact that these approaches have led to the discovery of cross-user vulnerabilities, they do not specifically target them and are inapt at completely modelling and detecting them. Moreover, although Android provides some guidelines on “Multi-User Support” in the Android Compatibility Definition Document (CDD)¹, there are no clear guidelines for framework developers on various access control aspects that should be carefully considered when implementing multi-user specific functionality [3].

Our goal. In this study, we aim to investigate thoroughly the security aspects regulating multi-user resource access, in an effort to derive static patterns used by APIs for enforcing access control. Once identified, we aim to use the patterns to evaluate OEM APIs as violations indicate potential multi-user specific vulnerabilities.

¹The Android Compatibility Definition Document (CDD) describes the requirements for device compatibility with the Android ecosystem, ensuring consistent behaviour across devices by OEMs.

Chapter 3

Motivation

In this chapter, we motivate our study by first discussing the impact of multi-user ecosystems in daily life. Then, we highlight the challenges vendors face in enforcing cross-user access control by providing motivating examples to illustrate these issues.

3.1 The Impact of Multi-User Android Ecosystems in Daily Life

Android’s popularity has driven the emergence of device-sharing opportunities enabled by the multi-user ecosystem. For example, Android used multi-user functionality as the foundation for implementing the Bring Your Own Device (BYOD) environment [34]. In workplaces, BYOD allows employees to use their personal devices for work through work profiles, which keep work apps separate from personal apps [2]. Additionally, Android Automotive, which runs directly on in-vehicle hardware, also supports multi-user functionality, allowing different users using the same vehicle to maintain separate profiles [1].

3.2 Problem: Why Are Vendors Failing to Protect APIs from Cross-User Attacks?

Robust cross-user access control is essential to prevent unauthorized access across different user profiles on the same device. However, our research reveals that vendors frequently

implement APIs that lack adequate protection mechanisms against cross-user attacks. Several factors contribute to this:

Reason 1: AOSP’s complex cross-user security specification is poorly documented causing vendors to unintentionally overlook them during customization resulting in little to no security in vendor APIs.

Reason 2: User access control checks exhibit considerable diversity and complexity, making it challenging to ensure comprehensive implementation. For instance, while one aspect of access control may be effectively implemented, other critical aspects may be inadvertently overlooked.

Reason 3: The implementation of user access control is performed through *patching* existing APIs, rather than having it considered in the design phase. This can naturally lead to vendors forgetting to patch certain code regions, hence leaving them vulnerable.

3.3 Motivation Example

To motivate the problem of missing cross-user access control checks, we present the following three APIs:

- **Listing 3.1:** `DevicePolicyManagerService.getStorageEncryption()` from AOSP source code
- **Listing 3.2:** `PersonaPolicyManagerService.getSecureFolderPolicy()` from Samsung Galaxy M20
- **Listing 3.3:** `VivoNightModeService.saveDarkModeStatusForMultiLand()` from Vivo Y33s

```

1 // DevicePolicyManagerService
2 public boolean getStorageEncryption(ComponentName who, int userHandle) {
3     if (!mHasFeature) {
4         return false;
5     }
6     checkArgumentNonnegative(userHandle, "Invalid userId");
7
8     final CallerIdentity caller = getCallerIdentity(who);
9
10    checkCallAuth(hasUserPerm(caller, userHandle));
11    ...
12    // API sink
13 }
14
15 private boolean hasUserPerm(CallerIdentity caller, int userHandle) {
16     return (userHandle == caller.getUserId()) ||
17         isSystemUid(caller) || isRootUid(caller) ||
18         hasPermission(INTERACT_ACROSS_USERS_FULL);
19 }

```

Listing 3.1: AOSP enforcing proper cross user access control checks

```

1 // PersonaPolicyManagerService
2 public List<String> getSecureFolderPolicy(String key, int userId){
3     List<String> list;
4     synchronized(this){
5         PersonalPolicyData policy = getPersonaData(userId);
6         list = policy.mSecureFolderPolicies.get(key);
7     }
8     return list;
9 }

```

Listing 3.2: Samsung API leaking secure folder policy data across users

```

1 // VivoNightModeManager
2 public void saveDarkModeStatusForMultiLand(String db, int st, int uid) {
3     long orig = Binder.clearCallingIdentity();
4     try {
5         try {
6             Settings.System.putIntForUser(getCR(), db, st, uid);
7         } catch (Exception e) {}
8     } finally {
9         Binder.restoreCallingIdentity(origId);
10    }
11 }

```

Listing 3.3: Vivo API enabling privileged setting changes by an under-privileged user

Listing 3.1 showcases a properly implemented cross-user access control check in the AOSP API `getStorageEncryption()`. We have redacted the code corresponding to actual functionality for brevity. In line 10, the API calls the `hasUserPerm()` method, which checks for cross-user permissions. These cross-user conditions are a disjunction of the following three checks:

1. The caller is invoking the API within its own context (line 16).
2. The caller is a privileged user (line 17).
3. The caller possesses the necessary system permission for cross-user API calls (`INTERACT_ACROSS_USER_FULL`) (line 18).

These checks safeguard the service API against unauthorized cross-user accesses.

On the other hand, [Listing 3.2](#) and [Listing 3.3](#) illustrate two custom APIs with improper multi-user access control implementation. (1) Samsung’s `getSecureFolderPolicy()` retrieves the secure folder policy for a specific user based on the `userId` parameter. This design allows a calling user to call this API and retrieve the secure folder policy information by simply supplying a `userId`. However, without access control checks on the `userId` parameter, a malicious user could exploit this to access other users’ secure folder policies by supplying their `userId`. (2) Vivo’s `saveDarkModeStatusForMultiLand()` allows the caller to update any setting within the system settings content provider for any user by providing the setting name, state, and `userId`. However, this API fails to enforce user-specific access control checks on the caller, allowing a non-privileged user to alter critical system settings for other users – a functionality that consistently requires a system privilege/cross-user permission check in AOSP APIs.

Our Study: In this study, we focus on detecting missing user-specific access control checks within the Android multi-user ecosystem. To achieve this, we systematically analyze over 2,500 APIs from the AOSP to characterize user-specific access control checks, identifying three distinct patterns, each designed to address a specific dimension of multi-user security. We then statically analyze custom system services and APIs to look for violations of these patterns, indicating potential multi-user vulnerabilities.

Chapter 4

Characterizing User Specific Access Control Checks

In this chapter, we perform an in-depth analysis of AOSP codebases to derive key features and operational mechanisms of the multi-user environment. Our analysis led to the identification of three comprehensive user-specific access control patterns.

4.1 Scope of Manual Analysis

To systematically investigate the intricacies of the multi-user system, we analyzed AOSP codebases versions 13 and 14 (corresponding to the latest versions at the time of writing). This involved code review of (a) 2032 APIS across 124 system services, (b) User-specific Broadcast Receivers, and (c) Lifecycle Methods defined within the analyzed system services.

4.2 Multi-User Access Control Patterns

Here, we describe the three multi-user access control patterns, identified through our manual investigation, highlighting their role in preventing cross-user vulnerabilities.

4.2.1 Pattern 1: User Status Management

To support multi-user implementation, Android provides mechanisms for user management, such as adding, removing, and switching users. It also ensures that user-related events pertaining to these operations (e.g., user switch, removal) are properly delivered to interested parties to maintain synchronous state management.

User status management requires dedicated access control enforcement at the level of Android APIs. Specifically, Android OS mandates that certain operations are permitted only for currently active users – i.e., foreground users interacting with the device. For instance, foreground users are exclusively allowed to access GPS coordinates, in order to prevent background users from inferring sensitive location information about active users. Generally, this restriction ensures that only the user actively engaging with the device has control over, and access to, critical system resources, thereby safeguarding user privacy and maintaining system integrity.

```
1 //UsbService
2 public void setUsbDeviceConnectionHandler(ComponentName handler) {
3     mContext.enforcePermission(MANAGE_USB, null);
4     synchronized (mLock) {
5         if (mCurrentUserId == UserHandle.getCallingUserId()) {
6             if (mHostManager != null) {
7                 setUsbDeviceConnectionHandler(handler);
8             }
9         } else {
10             throw new IllegalArgumentException();
11         }
12     }
13 }
```

Listing 4.1: AOSP APIs showcasing user status management patterns

We observe that this pattern is implemented by checking if the calling user of an API is equal to the current user. For example, [Listing 4.1](#) shows an AOSP API, `UsbService.setUsbDeviceConnectionHandler()`, that enforces that only the current user is allowed to set a USB device connection handler. Specifically, at line 5, the API checks if the calling user id (retrieved via `UserHandle.getCallingUserId()`) is equal to the global field `mCurrentUserId`, which reflects the currently active userId.

To correctly track currently active users (i.e., ensuring that global `userId` fields such as `mCurrentUserId` are updated correctly), we observe that system services perform the following three operations:

1. Initialize the global `userId` field:

Upon system bootup, Global `userId` fields must be initialized to reflect the first logged-in user. We observe that system services would typically set the global `userId` field to one of the following three values:

- **Current `userId`:** This is retrieved through `ActivityManager.getCurrentUser()`.
- **System `userId`:** This obtained through `UserHandle.USER_SYSTEM`.
- **Null `userId`:** Specifically, this corresponds to `UserHandle.USER_NULL`.

2. Update the global `userId` field to reflect the current user's `userId`:

System services should ensure that the global `userId` field is updated based on user-related system events. For example, in the event of a user switch, the services should update the field to reflect the newly switched user (otherwise, the field may reflect a background user).

```

1 // TvInputManagerService: User-related broadcast receiver
2 private void registerBroadcastReceivers() {
3     ...
4     IntentFilter intentFilter = new IntentFilter();
5     intentFilter.addAction(Intent.ACTION_USER_SWITCHED);
6     intentFilter.addAction(Intent.ACTION_USER_REMOVED);
7     intentFilter.addAction(Intent.ACTION_USER_STARTED);
8     intentFilter.addAction(Intent.ACTION_USER_STOPPED);
9     mContext.registerReceiverAsUser(new BroadcastReceiver() {
10         @Override
11         public void onReceive(Context context, Intent intent) {
12             String action = intent.getAction();
13             if(Intent.ACTION_USER_SWITCHED.equals(action)){
14                 switchUser(intent.getIntExtra(...));
15             }else if(Intent.ACTION_USER_REMOVED.equals(action)){
16                 removeUser(intent.getIntExtra(...));
17             }else if(Intent.ACTION_USER_STARTED.equals(action)){
18                 int userId = intent.getIntExtra(...);
19                 startUser(userId);
20             }else if(Intent.ACTION_USER_STOPPED.equals(action)){
21                 int userId = intent.getIntExtra(...);
22                 stopUser(userId);
23             }
24         }
25     }, UserHandle.ALL, intentFilter, null, null);
26 }
27
28 private void switchUser(int userId) {
29     synchronized (mLock) {
30         if (mCurrentUserId == userId) {
31             return;
32         }
33         ...
34         mCurrentUserId = userId;
35         ...
36     }
37 }

```

Listing 4.2: System services dynamically updating global `userId` fields through Broadcast Receiver

```

1 // PowerManagerService: User-related lifecycle method
2 public void onUserSwitching(@UserIdInt int newUserId) throws RemoteException {
3     synchronized (mLock) {
4         mUserId = newUserId;
5     }
6 }

```

Listing 4.3: System services dynamically updating global `userId` fields through Lifecycle Method

Android delivers user-related events to system services through dedicated system broadcasts and Lifecycle events. System services are expected to register to receive such events

and update the global `userId` fields accordingly.

Broadcast action example: For example, [Listing 4.2](#) shows a user-related Broadcast Receiver in `TvInputManagerService`. The Intent Filter in this code listens to a few user-related broadcast actions (lines 4-8), such as `ACTION_USER_SWITCHED` (line 5). Upon receiving this action (line 13), the method `switchUser()` (line 14) is triggered. It sets the global `userId` variable (line 34) to the parameter `userId` (supplied by the system), which denotes the newly switched `userId`. Similarly, the receiver handles other user-specific actions by updating the global `userId`s using the relevant functions.

Similarly, for other user-related events, the Broadcast Receiver updates the global `userId` using the relevant functions triggered within their respective control flow blocks.

Lifecycle event example: [Listing 4.3](#) shows the Lifecycle method `onUserSwitching()` defined in `PowerManagerService`, triggered by the system when a user switch event occurs. As shown, the method sets the global `userId` field `mUserId` (line 4) to reflect the second parameter `newUserId`, provided by the system.

3. Verify current `userId` against calling user's `userId`: In the last step of this access control pattern, APIs verify that the calling `userId` is equal to the current `userId` (maintained by the global `userId` field). This prevents background users from accessing resources and performing operations pertaining to the currently active user.

Special methods for retrieving the current `userId`. Besides relying on global `userId` fields, we observe that system services may directly invoke `ActivityManagerService.getCurrentUser()`¹. In such scenarios, the pattern implementation omits step 2 (i.e., updating the global `userId` field).

4.2.2 Pattern 2: User Ownership Enforcement

As mentioned earlier, Android implements isolation of users' resources (e.g., configuration, display preferences) and data (e.g., apps data) in order to ensure confidentiality and integrity. Specifically, users should not be able to access/modify another user's specific data/resources, as it could result in a loss of confidentiality or even harmful alterations.

¹Internally, `ActivityManagerService.getCurrentUser()` returns a `userId` field maintained in `UserController`.

```

1 //UserService
2 public boolean isUserNameSet(@UserIdInt int userId) {
3     final int callingUid = Binder.getCallingUid();
4     final int callingUserId = UserHandle.getUserId(callingUid);
5     if (!hasQueryOrCreateUsersPermission()
6         && !(callingUserId == userId
7             && hasPermission(GET_ACCOUNTS_PRIVILEGED, callingUid)))
8     {
9         throw new SecurityException();
10    }
11    synchronized (mUsersLock) {
12        final UserInfo userInfo = getUserInfoLU(userId);
13        return userInfo != null && userInfo.name != null;
14    }
15 }

```

Listing 4.4: AOSP API showcasing a user ownership check

Android APIs enforce such isolation between users through dedicated access control. Specifically, we observe that this pattern is implemented by checking if the calling user owns the resource they are trying to access. For example, [Listing 4.4](#) illustrates an AOSP API, `UserService.isUserNameSet()`, which determines whether a user has a username. In line 12, the API queries user information based on the `userId` supplied as an argument by the caller. To ensure that the caller can only query its own information (but not others), the API checks the supplied `userId` against the calling `userId` in line 6. If the two do not match, access is denied unless the caller holds other privileges (specifically, the permission `GET_ACCOUNTS_PRIVILEGED`).

To enforce user ownership over resources, system services perform the following operations:

- 1. Resolve calling user's `userId`:**

APIs first resolve the identity (i.e., `userId`) of the calling user process by invoking framework methods such as `Binder.getCallingUserId()` and `UserHandle.getUserId(Binder.getCallingUid())`.

- 2. Ensure calling user accesses its own resources:**

If an API allows operating on resources belonging to a user, whose `userId` is specified as a parameter to the API, the latter should ensure that the parameter matches the calling `userId`. Such ownership checks are often implemented in the form of conditional statements such as `Binder.getCallingUserid() == param`, where `param` is the `userId` supplied to the API by the caller.

Additionally, `UserController.handleIncomingUser()` method resolves the incoming `userId` based on the calling user's ownership rights. Therefore, APIs pass their supplied

userId parameter to `UserController.handleIncomingUser()` as an argument to resolve it using the following conditions sequentially, with the process halting at the first condition that is satisfied:

1. **System UID Check:** If the calling UID is `SYSTEM_UID`, the method returns the supplied userId without further checks.
2. **Caller's UserId:** If the supplied userId matches the calling userId, the method returns the supplied userId.
3. **Cross-User Permissions:** If the caller possesses cross-user permissions (`INTERACT_ACROSS_USERS` or `INTERACT_ACROSS_USERS_FULL`), the method returns the supplied userId.

4.2.3 Pattern 3: User Privilege Separation

Android APIs implement further restrictions based on the *privileges* associated with a given user. Specifically, a *System User*, who holds elevated privileges, can access a broader set of operations compared to *Secondary Users*. For example, a System user has the exclusive authority to add, delete, and modify other users and profiles.

Android assigns privileges to users based on the following three categories [10]:

1. **System User:** The most privileged user, with exclusive access to highly-sensitive settings and functionality.
2. **Secondary User:** A less privileged user created by the System User, who can perform less sensitive operations and is restricted from affecting other users (e.g., cannot add or delete users).
3. **Guest User:** A temporary user that operates with privileges equivalent to those of a Secondary User and is automatically deleted when it is no longer needed.

```

1 //UserService
2 public int[] getVisibleUsers() {
3     if (!hasPermission(INTERACT_ACROSS_USERS)) {
4         throw new SecurityException();
5     }
6     final long ident = Binder.clearCallingIdentity();
7     try {
8         return getVisibleUsers().toArray();
9     } finally {
10        Binder.restoreCallingIdentity(ident);
11    }
12 }

```

Listing 4.5: (A) AOSP API showcasing user privilege check

```

1 //CarActivityService
2 public int setPersistentActivity(ComponentName act, int id1, int id2) throws
3     RemoteException {
4     ensurePermission(Car.PERMISSION_CONTROL_CAR_APP_LAUNCH);
5     int caller = getCaller();
6     if (caller != UserManagerHelper.USER_SYSTEM &&
7         caller != ActivityManager.getCurrentUser())
8     {
9         return CarActivityManager.RESULT_INVALID_USER;
10    }
11    return setPersistentActivity(act, id1, id2);
12 }

```

Listing 4.6: (B) AOSP API showcasing user privilege check

We observe that this pattern is implemented by checking the privileges of the calling user before allowing access to certain sensitive resources/functionality. For example, the API `UserManagerService.getVisibleUsers()` in [Listing 4.5](#), ensures that only callers with the system permission `INTERACT_ACROSS_USERS` (line 3) can get a list of visible users. Similarly, the API `CarActivityService.setPersistentActivity()`, shown in [Listing 4.6](#), allows setting a persistent activity only if the calling user is a System User (or the current user) (line 4-5).

To enforce user privileges, system services can perform one of the following checks:

1. Ensure calling `userId` matches System `userId`:

The APIs first resolve the calling `userId` (similar to step 1 in Pattern 2), and then check if it is equal to `UserHandle.USER_SYSTEM`, which denotes the `userId` of the System User.

2. Ensure caller has certain system-level privileges: Here, instead of relying on the identity of the calling user, APIs can verify whether the calling process is itself privileged. For example, APIs may check whether the calling process (1) holds certain system-level permission such as `INTERACT_ACROSS_USERS` and `INTERACT_ACROSS_USERS_FULL`, or (2) belongs to the System (via `(getCallingUid() == Process.SYSTEM_UID)`) or Root (via `(getCallingUid() == Process.ROOT_UID)`).

Chapter 5

Design and Implementation

In this chapter, we present the end-to-end design of our tool, MVP, as shown in [Fig. 5.1](#). The primary objective of MVP is to identify vulnerable service APIs resulting from missing user-related access control checks, using the patterns outlined in [chapter 4](#). The design of our tool consists of two main phases: *Collection* and *Analysis*. In the Collection phase, we gather key Android framework artifacts which are necessary to identify the aforementioned patterns. In the Analysis phase, we perform static analysis using the collected artifacts to detect OEM APIs violating the identified patterns.

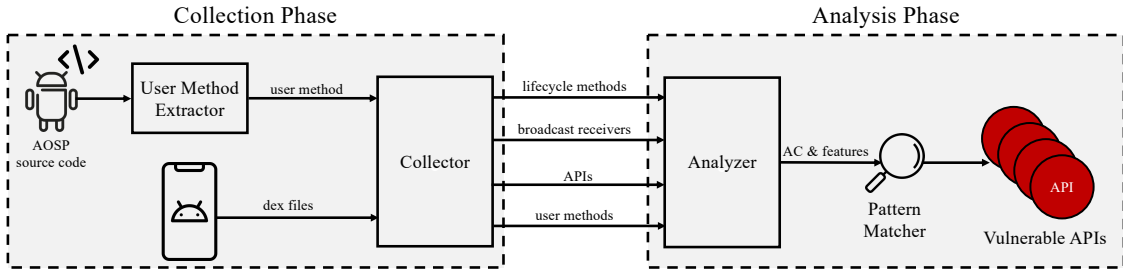


Figure 5.1: End-to-end design of MVP

5.1 Collection Phase

MVP ingests the OEM ROMs (i.e., `base/framework` classes) to extract Android framework artifacts. Next, we describe how each artifact is statically extracted to identify violations of user-specific access control checks.

Service API Collection. System services are registered through either `ServiceManager.addService()` or `ServiceManager.publishBinderService()`. To statically collect system services, MVP creates call graphs of all declared methods in the framework code-base and traverses the bytecode instructions to locate the invocation sites of the two aforementioned methods. Then, using the def-use chain, MVP resolves the arguments of the methods to retrieve registered service classes. Each service class implements an interface that declares all the service APIs. Using Class Hierarchy Analysis, MVP examines these interfaces to collect all public API entry points. By the end of the process, a service-to-API map is generated for the input Android image.

Broadcast Receiver Collection. Broadcast Receivers are registered through either `Context.registerReceiver()` or `Context.registerReceiverAsUser()`. MVP creates call graphs of all methods declared in the service classes and traverses its instructions to locate the invocation sites of these methods. Each method takes two arguments: an `IntentFilter` and a `BroadcastReceiver`.

MVP first resolves the `IntentFilter` object using def-use chain and tracks the invocation sites of all associated `addAction()` methods, which specify the action that the receiver can handle, such as `ACTION_USER_SWITCHED`. MVP then resolves the constant action string names. MVP further performs Class Hierarchy Analysis to locate the implementation of `onReceive()` method, which serves as the entry point for the Broadcast Receiver and defines its functionality.

Lifecycle Method Collection. Lifecycle methods are declared within a `Lifecycle` class, which typically extends a system service class. The `Lifecycle` class extends the abstract `SystemService` class and implements the Lifecycle Method callbacks.

To resolve Lifecycle Methods, we perform Class Hierarchy Analysis to first locate the `Lifecycle` class extending the system services. We then pinpoint methods implementing abstract methods in the `Lifecycle` base class. User-specific Lifecycle Methods are selected using a simple keyword matching (e.g., a method contains the keyword “User”).

User-related Method Collection. User-related methods are methods that either return a `userId` integer field or accept it as a parameter. These methods are a core part of our design for three reasons: 1) they enable us to identify access control checks, 2) they

help us infer whether a field used in a conditional statement is a `userId`, and 3) they allow us to extract global `userId` fields.

It is difficult to discern user-specific access control checks from non-user checks, as they can take various forms. In Android’s decompiled frameworks, parameter information and semantics are often missing, making it challenging to identify methods that perform user-related access control checks. To address this, we adopt a whitelisting approach, collecting AOSP-specific user-related methods from AOSP (where semantics are available due to access to the source code) that are common to all Android ROMs. Then, through static analysis, we discover vendor-specific user-related methods in custom ROMs where semantic information is unavailable.

[Listing 5.1](#) shows `onUserStarting()`; a Lifecycle Method from `SensoryPrivacyService`. At first sight, we cannot determine whether the global integer field `mCurrentUser` is a global `userId` field or not. However, In line 4, the global integer field `mCurrentUser` is assigned the return value of `UserHandle.getIdentifier()`, a user-related method that returns the `userId`. Since the return value of this user-related method is assigned to a global integer field within a system service, we can infer that `mCurrentUser` is a global `userId` field.

Similarly, [Listing 5.2](#) shows `getAccountsAsUser()`; an API from `AccountManagerService`. At first glance, we cannot establish with certainty whether `getAccountsAsUser()` operates on a `userId` and if so, which parameter it corresponds to. However, upon closer inspection of line 5, the API passes its second integer parameter, `param2`, as the second argument to `getAccountsAsUserForPackage()`, a user-related method that takes `userId` as the second parameter. From this, we infer that the `param2` parameter is a `userId` parameter.

```

1 //SensoryPrivacyService
2 public void onUserStarting(TargetUser user) {
3     if (mCurrentUser == USER_NULL) {
4         mCurrentUser = UserHandle.getUserIdentifier();
5         userSwitching(USER_NULL, getUserIdentifier());
6     }
7 }

```

Listing 5.1: (A) User-related method that facilitates the identification of `userId` fields and parameters

```

1 //AccountManagerService
2 public Account[] getAccountsAsUser(String param1, int param2, String param3) {
3     int callingUid = Binder.getCallingUid();
4     mAppOpsManager.checkPackage(callingUid, param3);
5     return getAccountsAsUserForPackage(param1, param2, param3, -1, callingPackage, false);
6 }

```

Listing 5.2: (B) User-related method that facilitates the identification of `userId` fields and parameters

We define two types of user-related methods:

1. **Type 1:** Methods Returning `UserId` Field
2. **Type 2:** Methods Taking a `UserId` as Parameter

To locate Type 1 and Type 2 user-related methods, we rely on semantic information in the AOSP source code – specifically, `userId`-related keywords and annotations. To this end, we process AOSP source code and extract method declarations using `JavaParser` for all methods [7] in the framework codebase. For each method declaration, we extract the parameter names, parameter types, return types, and annotations to pinpoint methods that return or operate on a `userId`.

We then expand this initial set of user-related methods by analyzing custom ROMs. Specifically, we perform static analysis to identify whether a given vendor method is user-related as follows:

- We resolve the return field of each vendor-introduced method using the def-use chain. If the return field matches a `userId` field identified via an AOSP method, we classify it as a Type 1 user-related method.

- We resolve the parameters of each vendor-introduced method using the def-use chain. If any parameter matches a `userId` field identified via an AOSP method, we classify it as a Type 2 user-related method.

Global `UserId` Field Collection. Here, we collect potential global `userId` fields, required to identify violations in User Status Management pattern (see § 4.2.1). As mentioned earlier, a correct user status check requires (1) a proper initialization of the variable global `userId`, and (2) ensuring that it is properly updated to reflect the current user.

To pinpoint global `userId` fields denoting the current user, we analyze all declared methods in the system service class. We create and traverse the CFG of these system methods to examine the following three types of instructions:

- **PUT instructions:** We resolve the value assigned to the global integer field. If it matches a system-defined `userId` constant (such as `UserHandle.USER_SYSTEM`) or a Type 1 user-related return value, we mark the field as global `userId`.
- **INVOKE instructions:** If an invoked method matches a Type 2 user-related method, we resolve the parameter at the `userId` position using def-use chains. If it is a global integer field, we mark it as a global `userId`.
- **CONDITIONAL BRANCH instructions:** We resolve the operands and predicates of conditional statements using def-use chains. If one of the predicates contains a `userId` constant or return value of a Type 1 user-related method and the other predicate is a global integer field, we mark the global integer field as a global `userId`.

For system services containing at least one `userId` field, we statically collect their initialization values and potential update call sites. To determine global `userId` initialization values, we resolve PUT instructions (associated with global `userId`) using def-use chains in system service constructors and boot-phase Lifecycle Methods. To identify potential update sites, we statically analyze Lifecycle Methods and the `onReceive()` method of user-related Broadcast Receivers to find internal methods updating the global `userId`.

Service API `UserId` Parameters. Here, we aim to pinpoint parameters denoting the `userId`, as it is required to identify violations of Pattern 2 (User Ownership Enforcement § 4.2.2) and Pattern 3 (User Privilege Separation § 4.2.3). We follow our methodology for identifying type-2 user-related methods in order to locate integer parameters that denote `userId`s. We omit details for brevity.

Access Control Check	Description
Permissions	Checks if the calling process has the required permissions to access or modify resources
UserId	Verifies the <code>userId</code> of the calling/current user to ensure it is authorized to perform the requested operation
UserId Resolution	Processes the <code>userId</code> using the <code>UserController.handleIncomingUser()</code> according to calling user's ownership rights
UID	Verifies the unique identifier (UID) of the process making the API call to ensure it is authorized to perform the requested operation
User Info	Utilizes properties in <code>UserInfo</code> to check if the calling user is authorized to execute the given operation

Table 5.1: Access Control checks extracted by the Analyzer Module of MVP

5.2 Analysis Phase

In this phase, MVP analyzes custom OEM APIs to extract implemented access control checks (summarized in [Table 5.1](#)). It then leverages the checks and framework artifacts (extracted in [§ 5.1](#)) to detect violations of the access control patterns, discussed [chapter 4](#).

5.2.1 Extracting Access Control Checks

To extract access checks for a given API, MVP first creates its CFG and traverses it in a DFS fashion. It then identifies *conditional branch* instructions and processes them to infer user-specific access control checks (e.g., one operand in the predicate evaluates to an *invoke* instruction of `UserHandle.getUserId(...)`) and extracts other constraints using def-use chains (e.g., the other operand constant value). It further identifies (non-conditional) statements used by Android to enforce permissions (e.g., an *invoke* instruction of `enforceCallingPermission(...)`) and similarly extracts corresponding arguments.

5.2.2 Detecting Pattern 1

MVP first determines if an API operates on a resource belonging to the current user. Specifically, it checks if any of the reachable instructions (apart from access control statements) is related to the global `userId` (e.g., an *invoke* instruction takes as an input the global `userId`). MVP then flags violations of Pattern 1 if the API misses to implement a current user check (i.e., misses to check if the calling user equals the global `userId`).

5.2.3 Detecting Pattern 2

Here, MVP proceeds similarly to detecting Pattern 1, with the exception that it targets APIs operating on a `userId` that is supplied as a parameter (i.e., the API allows accessing

resources belonging to a different user). In such cases, MVP looks for access control implementing ownership checks, and flags cases missing them. Namely, it flags APIs that fail to check if the supplied parameter is equal to the calling `userId`.

5.2.4 Detecting Pattern 3

Identifying Pattern 3 is more challenging as it requires inferring whether an API's functionality is highly-sensitive. Here, MVP relies on the following rules to determine if a reachable instruction in an API is (potentially) highly-sensitive: (1) an *invoke* instruction of an Android API requiring a system/signature permission (we use available API to permission maps to compile a list of highly-sensitive APIs), (2) an instruction that operates on a different `userId` (i.e., API allows accessing resource belonging to different users), and (3) a write operation to highly-sensitive content providers (e.g., `Setting` and `System` providers).

MVP then flags violations of Pattern 3 if the identified sensitive API misses to check that the calling user is a System User or that the calling process is highly-privileged (e.g., runs under `SYSTEM_UID`, or holds one of the user-related system permission checks).

Chapter 6

Evaluation

In this chapter, we answer the following research questions:

- **RQ1:** How prevalent are user-specific access control checks in AOSP and vendor ROMs?
- **RQ2:** To what extent do vendors increase the cross-user attack surface?
- **RQ3:** How do cross-user vulnerabilities evolve over time?

At the end of the section, we present two case studies: one involving the leakage of user data across different user profiles and the other allowing unauthorized manipulation of system settings by a non-privileged user.

6.1 Analysis Complexity and Configuration

We built MVP on top of WALA, a static analysis library for Java bytecode [59]. To construct the call graphs, we used `makeVanillaZeroOneCFABuilder` with a traversal depth of 5, balancing between capturing framework implementation details and preventing path explosion.

MVP takes, on average, approximately 40 minutes to analyze a single Android ROM. We parallelized MVP on 4 threads to optimize its runtime to 10 minutes per ROM.

All experiments were performed on an IBM Power LC922 server machine, which is equipped with a 22-core CPU (2.6 GHz POWER9 processor) and 256 GB of main memory.

6.2 Dataset Collection

For this study, we collected 73 Android ROMs, including 72 vendor ROMs and 1 AOSP ROM. While most ROMs were sourced from online repositories[28, 50, 11, 12], we also included ROMs from 3 physical devices for PoC testing (see § 6.4). Our dataset includes ROMs spanning Android versions 10 through 13 from AOSP and five popular vendors: Samsung, Vivo, Xiaomi, Oppo, and Tecno.

From the physical devices, we pulled the framework files using Android Debug Bridge [4]. For the ROMs downloaded from online repositories, we first extracted the `system.img` partition [17, 15, 56] and mounted it [13]. Once mounted, we extracted the framework dex files and decompiled them [23, 41, 40].

6.3 RQ1: How prevalent are user-specific access control checks in AOSP and vendor ROMs?

Vendor	Avg. APIs	Avg. Current UserId Checks	Avg. Calling UserId Checks	Avg. System UserId Checks	Avg. User Manager/ Info Checks	Avg. UID Checks	Avg. Permission Checks	Avg. UserId Resolution Checks
AOSP	2033	76	864	416	202	271	1172	181
Xiaomi	351	1	53	22	2	38	84	2
Oppo	331	1	3	1	0	13	101	0
Samsung	2166	43	494	143	143	205	513	5
Vivo	1054	4	53	6	4	18	232	5

Table 6.1: Average Counts of APIs and User-specific Access Control Checks per ROM Across Vendors

Our study reveals the prevalence of user-specific access control checks in both AOSP and vendor ROMs. We analyzed 51,667 APIs across our ROM corpus and identified 23,176 APIs (44.9%) enforcing at least one type of user-specific access control check. [Table 6.1](#) shows the average number of each type of user-specific access control check per ROM for both AOSP and vendors. It is important to note that for vendor ROMs, the statistics reflect access control enforcement in their custom APIs only.

User Status Management: Current userId checks are prevalent in AOSP, averaging 76 checks per ROM, which constitutes 3.7% of its total APIs. Samsung exhibits a notable frequency, with an average of 43 APIs implementing current userId checks, while Xiaomi, Oppo, and Vivo implement significantly fewer, with just 1, 1, and 4 checks, respectively. Across all vendors, a total of 49 checks are implemented, meaning 1.1% of custom APIs enforce current userId checks.

User Ownership Enforcement: Calling userId checks, User Manager/Info checks, and userId Resolution checks form the basis of user ownership enforcement. In AOSP, there are 864 Calling userId checks (42.5%), 202 User Manager/Info checks (9.9%), and 181 userId Resolution checks (8.9%). Samsung implements 494 Calling userId checks (22.8%), 143 User Manager/Info checks (6.6%), and 5 userId Resolution checks (0.2%). Xiaomi has 53, 2, and 2 checks for these categories, while Vivo implements 53, 4, and 5 checks respectively. Oppo, however, only has 3 Calling userId checks and no checks for the other categories, reflecting minimal inclusion of these checks in their system.

User Privilege Separation: System `userId` checks, UID checks, and Permission checks together form the basis of user privilege separation. AOSP enforces these checks the most, with 416 System `userId` checks (20.5%), 271 UID checks (13.3%), and 1,172 Permission checks (57.6%) per ROM. Samsung follows with 143 System `userId` checks (6.6%), 205 UID checks (9.5%), and 513 Permission checks (23.7%). In contrast, Xiaomi, Oppo, and Vivo show much lower enforcement, averaging 22, 38, and 84 checks for Xiaomi, 1, 13, and 101 for Oppo, and 6, 18, and 232 for Vivo, respectively.

Summary: User-specific access control checks are prevalent in vendor ROMs, with enforcement ranging from 119 checks in Oppo to 1,546 checks in Samsung (across the analyzed ROMs).

6.4 RQ2: To what extent do vendors increase the cross-user attack surface?

Vendor		v.10	v.11	v.12	v.13
Samsung	Pattern 1	1	1	0	0
	Pattern 2	17	10	10	10
	Pattern 3	0	0	0	0
Oppo	Pattern 1	0	0	0	0
	Pattern 2	0	0	3	3
	Pattern 3	0	0	1	1
Xiaomi	Pattern 1	0	0	0	0
	Pattern 2	2	2	2	0
	Pattern 3	0	0	0	0
Vivo	Pattern 1	0	0	0	0
	Pattern 2	0	0	1	1
	Pattern 3	1	1	1	1

Table 6.2: Access Control Pattern Counts by Vendor for Android Versions 10 to 13

Primary Breakdown:

MVP detects 69 total violations of user-specific access control check patterns, as shown in [Table 6.2](#)

The majority of these cases (88.4%) correspond to violations of the User Ownership Enforcement pattern. The rest 11.6% pertain to violations of User Status Management and User Privilege Separation patterns. This highlights the prevalence of user-related access control vulnerabilities in vendor ROMs.

Secondary Breakdown:

Here, we conduct a focused evaluation of the three custom ROMs, for which we had a physical device. We obtained a list of APIs flagged by MVP as vulnerable, and tested them to accurately estimate their outcome (we wrote a third-party app to trigger them).

In this experiment, we categorize the APIs as follows:

1. **Safe:** APIs that could not be invoked because of an enforcement of SEAndroid policy¹.

¹SEAndroid, based on Security-Enhanced Linux (SELinux), enforces mandatory access control (MAC) over all processes, including those with root privileges, to enhance system security, protect application data, and mitigate the impact of malicious software.[9]

Device	Service	API	Safe	Triggerable
Samsung Galaxy S20 FE 5G	CameraServiceWorker	getDeviceInjectorOverride	•	
	SemClipboardService	hasPrimaryClip	•	
	CredentialManagerService	getKeyguardStorageForCurrentUser		•
	NotificationManagerService	removeWearableAppFromList	•	
	NotificationManagerService	addWearableAppToList	•	
	OverlayManagerService	updatePackageCache	•	
	KnoxVpnEngineService	checkIfVendorCreatedKnoxProfile		•
	UniversalCredentialManagerService	getAdminForEnforcedCredentialStorageAsUser		•
	PersonaManagerService	getContainerOrder		•
	LauncherAppsService	changePackageIcon		•
	PowerManagerService	getLastUserActivityTime		•
	KnoxMUMRCPolicyService	sendRCPPolicyChangeBroadcast	•	
Oppo Reno 11 5G	AccessibilityManagerService	semPerformAccessibilityButtonClick	•	
	SecurityPermissionService	updateCachedPermission		•
	SecurityPermissionService	queryPackagePermissionsAsUser		•
Vivo Y33s	SecurityPermissionService	queryPermissionAsUser		•
	AutoBootService	changeAppAutoBootStatus	•	
	VivoNightModeService	saveDarkModeStatusForMultiLand		•
	AccessibilityManagerService	fixAccessibilityInputFilter	•	

Table 6.3: Corpus of System Service APIs missing user-specific access control checks, classified through manual testing into two categories: Safe and Triggerable

2. **Triggerable:** APIs that could be invoked successfully from a third-party app.

In total, we were able to trigger 10 (52.6%) of the APIs marked as vulnerable by MVP in the three analyzed physical devices. Notably, all the vulnerable APIs identified in Oppo Reno 11 5G were triggerable through a third-party application.

Summary: Our findings indicate that vendors substantially increase the cross-user attack surface through widespread violations of user-specific access control patterns.

6.5 RQ3: How do cross-user vulnerabilities evolve over time?

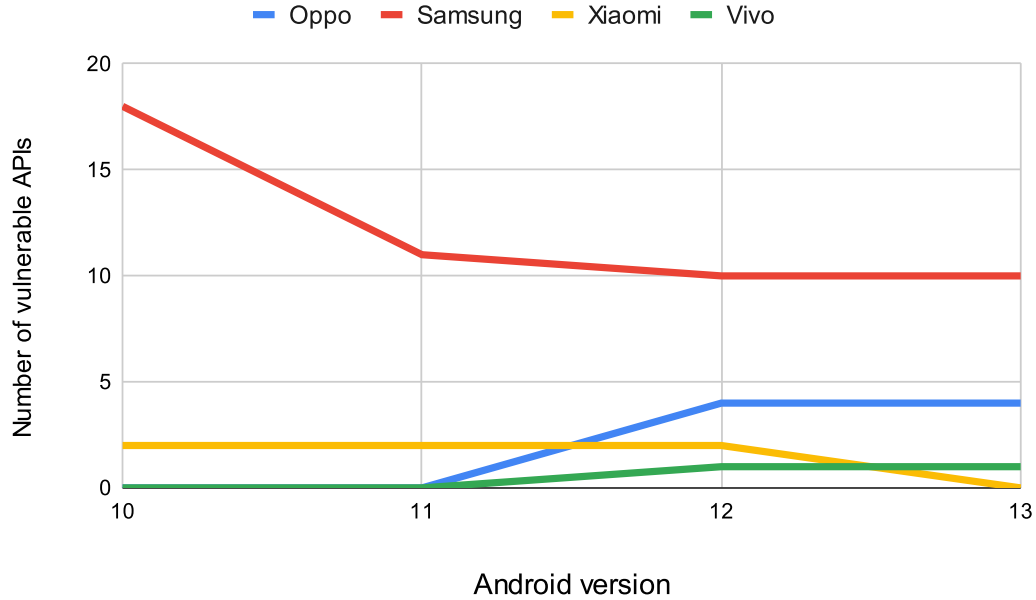


Figure 6.1: Number of vulnerable APIs missing user-related access control checks across different vendors and Android versions

We conduct a longitudinal study to analyze the evolution of vulnerable APIs for each OEM. Our study reveals that cross-user vulnerabilities follow varying trends across different OEMs, as shown in Fig. 6.1. For Samsung, the number of vulnerable APIs decreases significantly from Android version 10 to 11 and remains relatively stable thereafter. In contrast, Oppo shows an increasing trend, particularly between Android versions 11 and 12. Xiaomi’s vulnerabilities remain constant across versions 10, 11, and 12 before reducing to 0 in version 13. Vivo exhibits minimal vulnerabilities throughout the timeline.

Summary: Overall, the trends suggest that while some vendors, such as Samsung, have exhibited less cross-user vulnerabilities over time, others, like Oppo, have experienced an increase in vulnerabilities with newer Android versions.

6.6 Case Study 1: User Data Leak Across User Profiles

```
1 public long getLastUserActivityTime(int i) {  
2     Slog.d(TAG, "userId: " + i);  
3     if (mProfilePowerState.get(i) != null) {  
4         return mProfilePowerState.get(i).mLastUserActivityTime;  
5     }  
6     Slog.e(TAG, "error on userId:" + i);  
7     return -1L;  
8 }
```

Listing 6.1: `getLastUserActivityTime()` in `PowerManagerService`

In our investigation, we identified a data-leak vulnerability in the `getLastUserActivityTime()` API of the `PowerManagerService` class in a Samsung ROM (see [Listing 6.1](#)). This API is designed to return the last user activity time by querying an internal data structure containing profile power states based on the supplied `userId` (line 4). However, the API misses to enforce cross-user checks, allowing a user to illicitly access other users state information.

We confirmed the vulnerability on Galaxy S20 FE 5G and reported it to Samsung. The latter acknowledged and fixed the vulnerability.

6.7 Case Study 2: Privileged User Settings Updated by Non-privileged Users

```
1 public void saveDarkModeStatusForMultiLand(String db, int st, int uid) {  
2     long origId = Binder.clearCallingIdentity();  
3     try{  
4         try{  
5             Setting.System.putIntForUser(getCR(), db, st, uid);  
6             VLog.d("VNMS", "state=" + state + ",uid=" + uid);  
7         } catch (Exception e) {  
8             VLog.w("VNMS", "error=" + e);  
9         } finally {  
10            Binder.restoreCallingIdentity(origId);  
11        }  
12    }  
13 }
```

Listing 6.2: `saveDarkModeStatusForMultiLand()` in `VivoNightModeService`

In our investigation, we identified a privilege escalation vulnerability in the `saveDarkModeStatusForMultiLand()` API of the `VivoNightModeService` class in a Vivo ROM (see [Listing 6.2](#)). This API is designed to mutate the system setting of any arbitrary user. However, the API contains several flaws. Primarily, the third parameter, purported to be a UID, is improperly utilized as a `userId` and passed to the `Settings.System.putIntForUser()` method (line 5), without verifying that it matches the calling `userId`. Moreover, the API also provides parameters that enable passing the setting name (referred to as `db`) and state (referred to as `st`) to `putIntForUser()` invocation (line 5). Consequently, this API can be invoked with arbitrary parameters, enabling an underprivileged user to modify all System settings for a more privileged user.

We confirmed the vulnerability on Vivo Y33s device. Vivo has acknowledged and fixed the vulnerability.

Chapter 7

Related Works

7.1 Android Framework Multi-user

Researchers in the past have studied the multi-user ecosystem either comprehensively or as part of a broader analysis of Android framework access control enforcement.

Liu et al. introduced a user authority management model for Android, UAMdroid [43], designed to address the absence of multi-user management features in smartphone operating systems. UAMdroid extends the existing Android framework to allow users to share a device while performing user authority checks. This foundational work set the stage for later studies. Ratazzi et al.[48] provided a comprehensive assessment of Android’s multi-user framework, uncovering critical vulnerabilities such as privilege escalation and improper resource sharing between users. Their systematic approach identified key areas where access controls were either absent or incorrectly enforced, exposing significant security risks in the separation between users. The study not only highlighted these vulnerabilities but also designed experiments to test potential attack scenarios. Ratazzi et al. concluded that Android’s multi-user system, while offering basic user isolation, only partially mitigates the associated threats.

Our work builds on previous efforts, which are either outdated or lack sufficient focus, by systematically analyzing key components of the multi-user ecosystem. In our study, we focus on mapping the multi-user ecosystem through exploratory analysis of AOSP and identifying key Android artifacts that form the backbone of the multi-user ecosystem. Then, we use static analysis to identify instances of missing user-specific access control enforcement and detect cross-user vulnerabilities.

7.2 Inconsistent Access Control

The issue of inconsistent access control within the Android ecosystem has been extensively studied, revealing numerous vulnerabilities at both the application and framework levels. Due to a lack of standardization, researchers have focused on using automated and semi-automated approaches to mitigate security risks in the Android framework’s inconsistent access control enforcement.

Kratos [52] uses static analysis to construct a precise call graph, allowing it to perform inconsistency analysis to detect APIs leading to the same sink (e.g., internal method invocation) but enforcing different security checks. Building on this approach, AceDroid [18] advanced the methodology by modelling Android access control in a path-sensitive manner and normalizing diverse checks into a canonical form. Gorski [37] evaluated Android access control enforcement using semi-automated approaches, uncovering several multi-user vulnerabilities along with general inconsistency issues. ACMiner [36] used program and text analysis to find inconsistent access control enforcement, identifying 14 vulnerabilities in the Android multi-user ecosystem. Later, Gorski built ARF [38] on top of ACMiner to detect permission re-delegation vulnerabilities in framework APIs, identifying 3 vulnerabilities in the Android multi-user ecosystem. FReD [39] extended ARF’s design by combining Linux-layer file access control policies with Android-layer permission policies, leading to the detection of 2 additional multi-user vulnerabilities. IAceFinder [65], pioneered analysis of cross-context inconsistent access control enforcement in both Java and native layers. More recently, Poirot [30] has leveraged probabilistic inference to generate protection recommendations for Android framework resources. Bluebird [58] audits the protection mechanisms of Android framework APIs by combining app-side logic with statically derived program semantics using probabilistic inference.

Our work takes a targeted approach to analyze the multi-user ecosystem, identifying user-specific access control patterns in AOSP and detecting APIs that violate these patterns, which increases the cross-user attack surface.

7.3 Permission Maps

Research into Android’s permission framework has progressively advanced, offering deeper insights into Android API permission specifications.

Stowaway [35] laid the foundation for understanding Android’s permission system by using feedback-directed testing to compare the permissions requested by an application

with the maximum set actually required. In contrast to dynamic analysis, Pscout [24] uses static analysis to perform reachability analysis and list the permissions required by each API. Backes et al. [25] highlight the challenges of previous approaches and present Explorer, a tool that builds a static runtime model of the framework to improve the precision of previous works. To further improve the permission map specification, Aafer et al. propose Arcade [19], a static analysis tool that performs path-sensitive analysis using novel graph extraction techniques. Dynamo, on the other hand, takes a different approach by creating permission requirements for APIs through dynamic testing and fuzzing [27].

Our work statically analyzes service APIs to map a comprehensive feature set, including user-specific access control checks and artifacts related to multi-user access control enforcement. Using this feature set, we evaluate the multi-user ecosystem to identify cross-user vulnerabilities.

7.4 OEM Customization

Security vulnerabilities introduced through OEM customization of Android have been extensively studied across various layers of the system.

Wu et al. [61] analyzed 10 ROMs through permission-usage analysis to detect permission re-delegation and content leak vulnerabilities caused by due to vendor customization. In subsequent work, Aafer et al. [20] analyzed 591 ROMs and performed differential analysis to identify altered security features that introduce potential risks. Invetter [63] analyzes fragmented and poorly defined input validation checks using machine learning and static analysis, which led to 20 exploitable vulnerabilities in vendor ROMs. El-Rewini et al. [29] studied unused custom APIs and conducted a large-scale study to identify access control vulnerabilities caused by them. ADDITICTED [66] performs dynamic analysis to detect security hazards caused by customization of Android device drivers. SEPAL [62] investigates customizations in SEAndroid policies and identifies instances of unregulated rules using NLP techniques. Possemato et al. [46] conduct a longitudinal analysis to identify compliance and security issues caused by customizations in Android ROMs.

Our work conducts a longitudinal study to examine 73 ROMs and infer the prevalence of user-specific access control vulnerabilities caused by customization.

Chapter 8

Limitations

In this chapter, we discuss the limitations of our approach in finding missing user-related access control enforcements leading to potential cross-user attacks.

8.1 Vendors Enforcing Diverse Custom Access Control Checks

In this study, we focused on statically encoding the most common access control patterns found in AOSP and OEM ROMs. However, due to the diverse nature of vendor customizations, our approach might not capture all user-related access control mechanisms, especially those that are less frequently used or specific to certain vendors.

Vendor ROMs often implement custom mechanisms that deviate from standard AOSP patterns. These mechanisms may include OEM-specific access control checks, such as unique UIDs for privileged processes (e.g., Knox in Samsung devices). Such complexities present challenges for our technique, which is not fully aware of such variations.

Additionally, Android’s SEAndroid policy can restrict the invocation of seemingly-vulnerable system service APIs. We leave addressing SEAndroid policies to future work.

8.2 Missing Ground Truth

The multi-user ecosystem of the Android framework has not been extensively studied, leading to a lack of comprehensive understanding and standardization in detecting missing

user-related access control checks. Despite the growing importance of securing multi-user environments, there is no established ground truth against which current access control mechanisms can be evaluated. This gap presents a significant challenge in ensuring the robustness of user-related access control checks, as the absence of a standardized reference makes it difficult to systematically identify and address security vulnerabilities.

To address this gap, we conducted an independent ground truth study using a Samsung M20 ROM with Android version 10. Our analysis involved a detailed examination of 194 system services and over 2400 APIs. Through this manual inspection, we identified and marked APIs that were incorrectly flagged by MVP as missing user-related access control checks. The incorrectly marked APIs constituted less than 2% of the total examined. The primary reasons for these incorrect markings included missing API implementations, improper usage of the `userId` parameter, and cases where the global `userId` was not properly tracked.

Chapter 9

Conclusion

This thesis presents a thorough investigation of the Android multi-user ecosystem, with a focus on identifying vulnerabilities in user-specific access control mechanisms. Through systematic analysis, we identified key user-specific access control patterns and uncovered cross-user vulnerabilities across different vendors and Android versions. Our approach led to the discovery of several security gaps, emphasizing the ongoing need for rigorous enforcement of access control in multi-user environments.

The findings of this study contribute to a deeper understanding of the challenges associated with securing Android’s multi-user system, offering insights and motivation to the Android security community to provide better mechanisms for securing the multi-user framework.

References

- [1] Android automotive multi-user. <https://source.android.com/docs/devices/admin/multi-user#automotive-multi-user>.
- [2] Android bring your own device. https://static.googleusercontent.com/media/www.android.com/en//static/2016/pdfs/enterprise/Android_Enterprise_One_Pager_BYOD.pdf.
- [3] Android compatibility definition document multi-user support. https://source.android.com/docs/compatibility/14/android-14-cdd#95_multi-user_support.
- [4] Android debug bridge. <https://developer.android.com/tools/adb>.
- [5] Android manifest. https://android.googlesource.com/platform/frameworks/base/+/_master/core/res/AndroidManifest.xml.
- [6] Android settings. <https://developer.android.com/reference/android/provider/Settings>.
- [7] Javaparser. <https://javaparser.org/about.html>.
- [8] Multi-user apps. <https://source.android.com/docs/devices/admin/multiuser-apps>.
- [9] Security-enhanced linux in android. <https://source.android.com/docs/security/features/selinux>.
- [10] Supporting multi user. <https://source.android.com/docs/devices/admin/multi-user>.
- [11] Original samsung firmware updates, 2019.

- [12] Android dumps. <https://dumps.tadiphone.dev/dumps/>, 2024.
- [13] Imgtool. <https://newandroidbook.com/tools/imgtool.html>, 2024.
- [14] Knox: About shared devices. <https://docs.samsungknox.com/admin/knox-configure/how-to-guides/profiles/about-shared-device/>, 2024.
- [15] Lpunpack and lpmake, 2024.
- [16] Openai codex, 2024.
- [17] simg2img. <https://formulae.brew.sh/formula/simg2img>, 2024.
- [18] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *NDSS*, 2018.
- [19] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1164, 2018.
- [20] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting inconsistent security configurations in custom android {ROMs} via differential analysis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1153–1168, 2016.
- [21] Akinlolu Adekotujo, Adedoyin Odumabo, Ademola Adedokun, and Olukayode Aiyeniko. A comparative study of operating systems: Case of windows, unix, linux, mac, android and ios. *International Journal of Computer Applications*, 176(39):16–23, 2020.
- [22] Abdulla Aldoseri, David Oswald, and Robert Chipier. A tale of four gates: Privilege escalation and permission bypasses on android through app components. In *European Symposium on Research in Computer Security*, pages 233–251. Springer, 2022.
- [23] Anestisb. vdexextractor: Tool to decompile extract android dex bytecode from vdex files, 2024.
- [24] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

- [25] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Oteau, and Sebastian Weisgerber. On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*, pages 1101–1118, 2016.
- [26] Haipeng Cai and Barbara G Ryder. Understanding android application programming and security: A dynamic study. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 364–375. IEEE, 2017.
- [27] Abdallah Dawoud and Sven Bugiel. Bringing balance to the force: Dynamic analysis of the android application framework. *Bringing balance to the force: dynamic analysis of the android application framework*, 2021.
- [28] Firmware Drive, 2021.
- [29] Zeinab El-Rewini and Yousra Aafer. Dissecting residual apis in custom android roms. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1598–1611, 2021.
- [30] Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. Poirot: Probabilistically recommending protections for the android framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 937–950, 2022.
- [31] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android’s Security Architecture*. No Starch Press, USA, 1st edition, 2014.
- [32] 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 *29th USENIX security symposium (USENIX Security 20)*, pages 2379–2396, 2020.
- [33] William Enck, Damien Oteau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC’11, page 21, USA, 2011. USENIX Association.
- [34] Maman Fathurrohman, Hepsi Nindiasari, Nurul Anriani, and Aan Subhan Pamungkas. Empowering mathematics teachers’ ict readiness with android applications for bring your own devices (byod) practice in education. *Cogent Education*, 8(1):2002131, 2021.

- [35] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.
- [36] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android’s middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 25–36, 2019.
- [37] Sigmund Albert Gorski III. *Semi-automated Evaluation of Access Control Enforcement in the Android Platform*. North Carolina State University, 2020.
- [38] Sigmund Albert Gorski III and William Enck. Arf: identifying re-delegation vulnerabilities in android system services. In *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*, pages 151–161, 2019.
- [39] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. {FReD}: Identifying file {Re-Delegation} in android system services. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1525–1542, 2022.
- [40] iBotPeaches. Apktool: A tool for reverse engineering android apk files, 2024.
- [41] JesusFreke. Smali: Smali/baksmali, 2024.
- [42] Rui Li, Wenrui Diao, Zhou Li, Jianqi Du, and Shanqing Guo. Android custom permissions demystified: From privilege escalation to design shortcomings. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 70–86. IEEE, 2021.
- [43] Zemin Liu, Choon-Sung Nam, and Dong-Ryeol Shin. Uamdroid: A user authority manager model for the android platform. In *13th International Conference on Advanced Communication Technology (ICACT2011)*, pages 1146–1150, 2011.
- [44] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. Trellis: Privilege separation for multi-user applications made easy. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*, pages 437–456. Springer, 2016.
- [45] Hashida Haidros Rahima Manzil and S Manohar Naik. Detection approaches for android malware: Taxonomy and review analysis. *Expert Systems with Applications*, page 122255, 2023.

- [46] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *2021 IEEE symposium on security and privacy (SP)*, pages 87–102. IEEE, 2021.
- [47] Swapnil Powar and BB Meshram. Survey on android security framework. *International Journal of Engineering Research and Applications*, 3(2):907–911, 2013.
- [48] Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. A systematic security evaluation of android’s multi-user framework, 2014.
- [49] Martin P Robillard and Gail C Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.
- [50] SamMobile. Download firmware updates for your samsung mobile phone and tablet, Aug 2019.
- [51] Oliver Schranz, Sebastian Weisgerber, Erik Derr, Michael Backes, and Sven Bugiel. Towards a principled approach for dynamic analysis of android’s middleware. *arXiv preprint arXiv:2110.05619*, 2021.
- [52] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.
- [53] steadfasterX. Steadfasterx/salt: Salt - [s]teadfasterx [a]ll-in-one [l]g [t]ool, 2024.
- [54] Davi Tabosa, Oton Pinheiro, Lincoln Rocha, and Windson Viana. A dataset of atoms of confusion in the android open source project. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 520–524, 2024.
- [55] Mohammed El Amin Tebib, Pascal André, Mariem Graa, et al. Privdroid: Android security code smells tool for privilege escalation prevention. In *2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCCom/CyberSciTech)*, pages 1024–1031. IEEE, 2023.
- [56] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Ruales, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. Attention spanned: Comprehensive vulnerability

analysis of AT commands within the android ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

- [57] Pradeep Kumar Tiwari and Upasna Singh. Android users security via permission based analysis. In *Security in Computing and Communications: Third International Symposium, SSCC 2015, Kochi, India, August 10-13, 2015. Proceedings 3*, pages 496–505. Springer, 2015.
- [58] Parjanya Vyas, Asim Waheed, Yousra Aafer, and N Asokan. Auditing framework {APIs} via inferred app-side security specifications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6061–6077, 2023.
- [59] Wala. Wala/wala: T.j. watson libraries for analysis, with frontends for java, android, and javascript, and may common static program analyses, 2024.
- [60] Haoyu Wang, Yao Guo, Zihao Tang, Guangdong Bai, and Xiangqun Chen. Reevaluating android permission gaps with static and dynamic analysis. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2015.
- [61] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634, 2013.
- [62] Dongsong Yu, Guangliang Yang, Guozhu Meng, Xiaorui Gong, Xiu Zhang, Xiaobo Xiang, Xiaoyu Wang, Yue Jiang, Kai Chen, Wei Zou, Wenke Lee, and Wenchang Shi. SEPAL: towards a large-scale analysis of seandroid policy customization. *CoRR*, abs/2102.09764, 2021.
- [63] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating insecure input validations in android services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1178, 2018.
- [64] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Droidalarm: an all-sided static analysis tool for android privilege-escalation malware. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 353–358, 2013.
- [65] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. In *The 2022 Network and Distributed System Security Symposium (NDSS’22)*, 2022.

- [66] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, page 409–423, USA, 2014. IEEE Computer Society.
- [67] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.