

# A Longitudinal Analysis Of Replicas in the Wild Wild Android

by

Syeda Mashal Abbas Zaidi

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

© Syeda Mashal Abbas Zaidi 2024

## **Author's Declaration**

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

## **Statement of Contributions**

This thesis is based on a paper I co-authored with Shahpar Khan, Parjanya Vyas and Prof. Yousra Aafer. The paper is accepted for publication at ASE 2024. I developed the Path Extraction, Semantic Similarity, and the Security Audit module for the tool we develop in the study.

## Abstract

In this thesis, we report and study a phenomenon that contributes to Android API sprawls. We observe that OEM developers introduce private APIs that are composed by copy-paste-editing full or partial code from AOSP and other OEM APIs – we call such APIs, Replicas.

To quantify the prevalence of Replicas in the wildly fragmented Android ecosystem, we perform the first large-scale (security) measurement study, aiming at detecting and evaluating Replicas across 342 ROMs, manufactured by 10 vendors and spanning 7 versions. Our study is motivated by the intuition that Replicas contribute to the production of bloated custom Android codebases, add to the complexity of the Android access control mechanism and updates process, and hence may lead to access control vulnerabilities.

Our study is facilitated by RepFinder, a tool we develop. It infers the core functionality of an API and detects syntactically and semantically similar APIs using static program paths. RepFinder reveals that Replicas are commonly introduced by OEMs and more importantly, they *unnecessarily* introduce security enforcement anomalies. Specifically, RepFinder reports an average of 141 Replicas per the studied ROMs, accounting for 9% to 17% of custom APIs – where 37% (on average) are identified as under-protected. Our study thus points to the urgent need to debloat Replicas.

## Acknowledgements

Firstly, I would like to thank God for providing me the opportunity, strength, and support to go through this research journey.

I would like to express my deepest gratitude to my supervisor, Prof. Yousra Aafer, for her unwavering support and invaluable guidance throughout the course of my thesis. I would also like to thank my readers Prof. Urs Hengartner and Prof. Meng Xu for their time and feedback.

I would like to thank Shahpar Khan and Parjanya Vyas for all their help and support throughout the project.

Most importantly, I am grateful to my family for their unwavering support and love. In particular, I want to thank my husband, Qaswar Hasnain, my parents, Jari and Sonia Abbas, and my sisters, Sania, Zahra, and Maryam. Their encouragement and belief in me have been a constant source of strength throughout this journey.

## **Dedication**

To my family.

# Table of Contents

<b>Author's Declaration</b>	<b>ii</b>
<b>Statement of Contributions</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>v</b>
<b>Dedication</b>	<b>vi</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>5</b>
2.1 Cloning Private APIs . . . . .	5
2.2 Can Cloning API Go Wrong? . . . . .	6
<b>3 How To Detect Replicas</b>	<b>8</b>
3.1 Replica Peculiarities . . . . .	8

<b>4</b>	<b>OUR APPROACH: RepFinder</b>	<b>11</b>
4.1	Paths Extraction . . . . .	11
4.2	Paths Filtering . . . . .	13
4.3	Core Paths Selection . . . . .	14
<b>5</b>	<b>Similarity detection</b>	<b>16</b>
5.1	Measuring Similarity . . . . .	16
5.2	Syntactic Similarity . . . . .	17
5.3	Measuring Semantic Similarity . . . . .	18
<b>6</b>	<b>Security Audit of Replicas</b>	<b>19</b>
6.1	Access Control Consistency . . . . .	19
6.2	Updates Propagation Consistency . . . . .	20
<b>7</b>	<b>Large Scale Measurement Study</b>	<b>21</b>
7.1	Dataset Collection . . . . .	21
7.2	Analysis Landscape and Complexity. . . . .	23
<b>8</b>	<b>Replicas Outlook</b>	<b>25</b>
8.1	RQ1. Replicas Prevalence . . . . .	25
8.2	RQ2. Addition, Removal, and Inheritance Trends . . . . .	28
8.3	RQ3. Replicas Security Audit Results . . . . .	30
<b>9</b>	<b>Comparison With other Tools</b>	<b>35</b>
<b>10</b>	<b>Threats to Validity</b>	<b>37</b>
10.1	Internal Threats to Validity . . . . .	37
10.2	External Threats to Validity . . . . .	37
<b>11</b>	<b>Discussion</b>	<b>38</b>

<b>12 Related Works</b>	<b>39</b>
12.1 Android Framework Analysis . . . . .	39
12.2 OEM-Customization Hazards . . . . .	40
12.3 Code Clone Detection . . . . .	41
<b>13 Conclusion</b>	<b>45</b>
<b>References</b>	<b>46</b>
<b>APPENDICES</b>	<b>56</b>
<b>A</b>	<b>57</b>
A.1 Our Solution . . . . .	57
A.2 Jaccard Similarity . . . . .	58

# List of Figures

2.1	AOSP's <code>monitorGestureInput</code> and its vulnerable ZTE Replica <code>myInput</code> . . . . .	6
3.1	Cloning in API implementations . . . . .	9
4.1	Workflow of RepFinder . . . . .	12
8.1	Replicas Breakdown . . . . .	26
8.2	Addition, Removal Trends, per Vendor . . . . .	29
8.3	Access Control Inconsistencies in Replicas . . . . .	30
8.4	Distribution of Access Control Checks . . . . .	31
8.5	Security Updates Consistency in Replicas . . . . .	32

# List of Tables

4.1	Paths identified in <code>removeData()</code> in <code>OplusDevicePolicy</code> – Oppo v.13 . . .	11
5.1	Transformation Rules . . . . .	18
7.1	ROM Dataset . . . . .	22
7.2	Traces and Embeddings For ROMs . . . . .	23
8.1	Impact of Similarity Threshold . . . . .	28
8.2	Vulnerabilities Manually Verified . . . . .	33
9.1	Clones Detected by Tools . . . . .	36
A.1	Paths identified in AOSP <code>getUidStats(..)</code> . . . . .	57
A.2	Paths identified in AOSP <code>getUidStatsWithoutCheckUid(..)</code> . . . . .	57

# Chapter 1

## Introduction

The practice of copy-paste-editing code snippets, a.k.a., code cloning has been widely adopted by Android developers. App developers copy code snippets from discussion forums (e.g., StackOverflow) and from open source code (app) repositories to quickly reuse already implemented functionality.

Research suggests [24, 89] that code cloning in Android apps carries risks. It contributes to the production of bloated app codebases, increasing both code complexity and maintenance efforts. From a security perspective, cloning is detrimental if not well carried out. Existing research shows that cloning causes the propagation of insecure and vulnerable code snippets from StackOverflow discussion forums into millions of Android apps [33]. Other research suggests that significant vulnerabilities are introduced in apps through cloned code (from app and library-specific code) [80, 85, 83, 20, 76].

Despite the numerous efforts [24, 17, 31, 91, 89] on detecting code clones in the Android app layer, little has been done to understand the extent of code duplication in the Android framework itself, not to mention any efforts to assess its security implications. In this thesis, we fill the gap by conducting the *first large-scale* study aimed at a better understanding of the practice of code duplication in the Android framework during the customization process. Our study particularly focuses on identifying duplicate APIs — that is, exposed framework entry points that allow apps to access system resources. We call duplicate APIs *Replicas*.

Many approaches have been proposed to detect code clones. They can be broadly categorized into three categories, token-based [36, 42, 49, 53, 71, 59], structural-based [70, 87, 92, 41, 77, 82], and deep-learning based techniques [41, 74, 77, 82]. In the context of Android apps, structural similarity (PDG-based [24, 33] and CFG-based [80]) is the

defacto approach for detecting similar apps. While these approaches are able to detect some Replicas, they are likely to lead to inaccurate results due to a number of shortcomings. Prior static approaches often assume that all code regions are *equally important* to the core functionality, and hence are all factored in when computing the similarity score.

However, this assumption is often incorrect in Android APIs, which may lead to significant false positives. We observe that when OEM developers introduce Replicas, they focus on copying (and refactoring) core-logic functionality and pay less attention to other logic (e.g., they may miss input validation, access control and error handling logic). Therefore, the direct application of existing code clone detection solutions for Replica identification will likely lead to both false alarms and false negatives. Specifically, as they cannot point out core functionality, the tools will likely flag APIs sharing non-core logic as Replicas (i.e., false alarms), and may miss to detect true Replicas when two APIs diverge on significant non-core logic.

**RepFinder.** To address these limitations, we developed *RepFinder*, a new analysis pipeline for automatically detecting Replicas. We argue that path-sensitivity is crucial for summarizing and pointing the core functionality of Android APIs, hence RepFinder models APIs in the form of *static program paths*. Specifically, RepFinder matches two APIs by capturing similarity of distinct paths (i.e., ordered sequence of instructions) that start at an API's entry point and end at a (non-error) termination point. To avoid inaccuracies of prior tools, RepFinder leverages program analysis and NLP analysis to identify and limit the detection scope to *core program paths*.

Android Replicas span the traditional clone types; Type-1 to Type-4 (i.e., exact copies, near-miss clones, and semantic clones) [23]. Thus, we introduce similarity detection measures to capture the degree of similarity between core program paths across APIs. The measures allow calculating syntactic and semantic similarity.

Conceptually, Type 1-3 clones can be detected by a semantic similarity measure. However, we opt to use an approach that combines syntactic and semantic similarity calculation for scalability concerns. Although limited to detecting Types 1-3 Replicas, syntactic similarity measure is efficient and scalable as it involves fast set-based comparison – hence, it can perform an exhaustive pair-wise comparison of all custom APIs in a ROM efficiently. The semantic detection measure, on the other hand, can effectively detect Type-4 clones; however, we observe that it is substantially slower and thus cannot be applied on large scale. Our experimentation reveals that it takes 10 hours to perform a semantic pair-wise API comparison on an average size ROM. Our approach therefore combines the two measures as follows: it conducts syntactic similarity detection across all APIs in a ROM, and exclusively conducts semantic similarity detection on APIs defined within the same system

service.

**Large measurement study.** To conduct the large-scale (security) measurement study of Replicas in the Android ecosystem, we collect a dataset of 342 ROMs, spanning 7 versions and 10 vendors – including big players in the Android ecosystem such as Samsung, Xiaomi, and Oppo. Altogether, RepFinder analyzed 44,794 APIs and 6,248,226 program paths.

To understand and quantify the security impact of Replicas, we focus on access control enforcement with regard to two aspects. First, we evaluate the adequacy of access control implementation adopted in a Replica by comparing it against that of the *original* API (from which the Replica was cloned). Since the two APIs achieve the same functionality, they should enforce similar access control. A weaker or absent access control will inevitably introduce vulnerabilities. In such scenarios, third-party apps can invoke the Replica to access the functionality, without satisfying the privilege requirement in the original API.

Second, we check the consistency of access-control updates for each pair of <Original, Replica> across major versions. Intuitively, Replicas add complexity to the already-complicated Android updates procedure and require OEM developers to keep track of replicated functionality. Specifically, during the update procedure, OEMs should be able to identify pairs of <Original, Replica> and ensure that updates to the original API are properly propagated to its Replica. Unfortunately, this task is challenging since Replicas may exist in an undocumented state (i.e., unknown to OEM developers).

**Key findings.** Our study is the first of its kind to detect Replicas in the wildly fragmented Android ecosystem. In the collected dataset, RepFinder reports an average of 141 Replica per ROM. The prevalence of Replicas varies across vendors – Ranging from 9% in Xiaomi and reaching up to 17% in Lenovo.

The study reveals a general decrease trend in the latest versions, which is mainly due to vendors removing *inherited* Replicas (in versions 13 onwards). However, the study also shows that vendors do introduce *new* Replicas in the latest versions. At times, the number of added Replicas outweighs removed Replicas. This clearly indicates that Replicas are not an issue of the past.

Most importantly, our study highlights the security risks associated with Replicas. The ratio of under-protected Replicas is high, averaging 37% and reaching up to 51%. Through analyzing the security updates propagation landscape in Replicas, we found that security patches are not properly propagated in 375 <Original, Replica> pairs; resulting at times, in more permissive Replicas than their original APIs.

To concretely understand the security impact of under-protected Replicas, we selected a few reports based on device availability and severity of missing checks. We exploited

7 different Replicas to achieve various attacks. We reported all findings to corresponding vendors. Notably, we were able to write a keylogger on ZTE, break Android’s app sandbox model (i.e., by writing to any random app directory (vendor A<sup>1</sup>) and by reading any random app directory (vendor A)), update Android’s Secure Settings and read a device’s Build serial Id on Tecno. We exploited other Replicas to achieve less critical but permission-protected functionality. At the time of writing, 3 CVEs were issued and 2 are under request. We are still awaiting reports from other vendors. Overall, our findings point to *unnecessary* security hazards introduced by improperly cloned AOSP code and the urgent need to debloat them.

We summarise our contributions as follows:

- We perform the first, large-scale, security study of Replicas in the wild fragmented Android ecosystem.
- We put forward RepFinder, an analysis pipeline combining program and NL-PL understanding, that is tailored to Android Replica characteristics, to automatically identify Replicas and audit their access control enforcement.

We make our code and sample ROMs available at the repository <https://zenodo.org/records/11521246>.

---

<sup>1</sup>masking the vendor’s name until the vulnerability is patched

# Chapter 2

## Background

In this section, we lay the necessary background on Android APIs cloning and discuss a vulnerable Replica to motivate our study.

### 2.1 Cloning Private APIs

Android OEMs, also known as vendors, tailor AOSP codebases for custom functionality by adding new private APIs. For example, a vendor might introduce a new private API to allow OEM developers to access a new camera hardware. Unlike public APIs, which are centrally managed by Google, OEM private APIs are under-regulated, often erratically added, altered, and removed by OEMs [62].

Although Android private APIs are not inherently problematic, they can lead to an *API sprawl*; where APIs uncontrollably proliferate within the Android ecosystem, mainly due to the lack of governance. Prior research suggests that APIs sprawls in Android lead to Residuals [29]: those that are no longer used but are still lurking around in the Android framework. The research further demonstrates that Residuals lead to vulnerabilities (e.g., deprecate security checks) and to anomalous access control enforcement.

In this work, we report a different phenomena behind API sprawls. We observe that some private APIs are composed by copy-paste-editing full or partial code from AOSP and other OEM APIs, resulting at times in duplicate functionality provided by different APIs. We refer to such duplicate APIs as Replicas we refer the reader to [section 2.2](#) for an example.

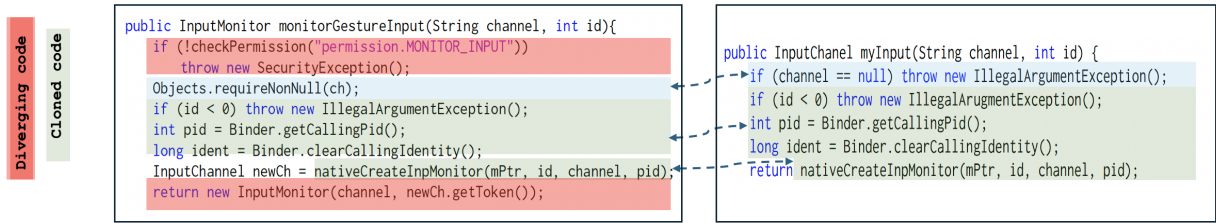


Figure 2.1: AOSP’s `monitorGestureInput` and its vulnerable ZTE Replica `myInput`

## 2.2 Can Cloning API Go Wrong?

In our analysis of Android APIs, we found that there are cloning scenarios that are legitimate. API cloning avoids risking the stability of AOSP APIs and preserves compatibility. For example, when introducing a variation of AOSP hardware, OEMs tend to implement new private API, by cloning the AOSP API instead of modifying it, as there are often non-trivial modifications required.

Despite these good intentions, cloning in Android, particularly at the app layer, has long been associated with different problems, including design, maintenance, and security issues [24, 17, 31, 91, 89]. At the framework layer, we argue that cloning APIs could similarly introduce serious issues. For example, as discussed later (section 3.1), cloning APIs mainly concerns core functionality, possibly leading to omitting security-critical code such as access control implementation. Besides, cloning may complicate the patching of APIs as it needs to be applied in multiple locations, often by different entities (e.g., Google and OEMs).

### *Motivating Vulnerable Replica:*

Figure 2.1 shows AOSP API `IInputManager.monitorGestureInput(..)` and its Replica `IInputManager.myInput(..)`, defined by ZTE in Axon A40. The two APIs implement highly identical *core functionality* with a slight variation. The shared code (in green) creates an input channel that receives input events (e.g., screen taps) from the input dispatcher. Essentially, the input channel enables tracking system-wide input events. The API `monitorGestureInput(..)` includes a few code lines (in red) that are missing from the ZTE Replica. The last line (in red) wraps the created channel in an input monitor. Observe the APIs perform a semantically-identical validation of the first input, using different syntax (in blue).

Given the sensitivity of the operation, the original API `monitorGestureInput(..)` enforces a system permission (`MONITOR.INPUT`). The check ensures that only qualified system

processes and system apps can track user input events. Unfortunately, ZTE Replica misses this critical check – indicating that the OEM developer was mainly concerned with copying the core logic functionality.

We exploited the Replica to write a keylogger, and reported it to ZTE. ZTE acknowledged and fixed the vulnerability in its latest models. A CVE id has been issued.

# Chapter 3

## How To Detect Replicas

We observe that Replicas span the four traditional categories of clone types [23] with specific peculiarities.

### 3.1 Replica Peculiarities

We use two Replicas detected in AOSP and Oppo to illustrate how API cloning is performed. The AOSP API `setPowerMode` and its Type-2 clone `setPowerModeChecked` in Figure 3.1(A) are syntactic Replicas – `setPowerModeChecked` implements identical functionality to `setPowerMode`, but additionally returns whether the call was successful. As the APIs differ only in the `return` statement, they will be easily identified as clones by existing approaches. While some Replicas may similarly follow the above easily detectable cloning pattern, we observe that other patterns are more challenging. The main difficulties are as follows:

#### **Challenge 1: The implementation of Replicas may be discrepant**

We observe that *the implementation of Replicas may include substantial differences, converging only on the core-logic functionality*. Figure 3.1(B) reports a divergent implementation of Oppo’s Replica `getUidStatsWithoutCheckUid` from its original AOSP API `getUidStats`. As shown, the two APIs implement identical core logic, depicted by the native method `nativeGetUidStat` taking similar arguments. However, the AOSP API `getUidStats` includes additional logic in the form of two disjunctive checks. The first is



Figure 3.1: Cloning in API implementations

an access control, enforcing the caller is a system process (i.e., `uid` equals 1000), while the second is an input validation, ensuring that the supplied `uid` matches the calling process’s `uid`. The API aborts execution if none of the checks is satisfied. We observe that this practice is common during API cloning. OEM developers focus on copying (and refactoring) core-logic functionality instead of other logic (i.e., they pay less attention and may miss input validation, access control logic, pre-condition checks, logging statements, etc). If we directly utilize existing clone detection methods to detect Replicas, they will achieve sub-optimal performance. Since they cannot distinguish core from non-core functionality, the methods will likely miss detecting true Replicas when they diverge on significant non-core functionality.

To illustrate this limitation, we use SourcererCC [59], a state-of-the-art token-based clone detection method. To calculate similarity of two APIs, SourcererCC divides the number of shared tokens of the two APIs by the maximum of the number of tokens of the two APIs to obtain the overlapping similarity. For the pair `getUIDStats` and `getUidStatsWithoutCheckUid`, the number of tokens is 34 and 9, respectively. The overlapping similarity is  $9/34 = 0.26$  (where 9 is the number of shared tokens). However, the default similarity threshold of SourcererCC is 0.7 and code pairs with a threshold below that are considered non-clones. Hence, SourcererCC fails to detect that the APIs *are true*

*Replicas.*

We note that this characteristic of Replicas, allows to narrow down its definition to the following: *Replicas are a pair of Android APIs that implements the same logical functionality, but may diverge on other code.*

## Challenge 2: Android (non-Replica) APIs share significant code

Android APIs, particularly those defined within the same system service, often share non-core functionality (e.g., same implementation of error handling, input validation, etc). At times, the non-core code outweighs the core functionality code, often leading to different (non-Replica) APIs having highly similar structural representation (e.g., large number of similar independent subgraphs in the case of PDG representation), or sharing significant tokens.

Figure 3.1(C) showcases an excerpt of AOSP APIs `hasBindAppWidgetPermission` and `setBindAppWidgetPermission`, defined in `AppWidgetService` system service, retrieved from AOSP codebase (version 14). As shown, the APIs are not Replicas since they implement different core functionality; the first checks if an app has the specified permission while the second grants (or revokes) that permission to (or from) an app. Yet, the APIs share a large number of tokens and semantic blocks<sup>1</sup>. Thus, the APIs will be falsely detected as clones both by existing token-based and PDG-based approaches.

To demonstrate this, we followed the methodology proposed by DNADroid [24] to calculate the similarity score of the two APIs based on their PDGs. The score obtained is 0.93 – as the score is higher than DNADroid’s similarity threshold (0.7), the APIs are falsely considered Replicas.

**Our Solution.** To address these challenges, we detect Replicas by *focusing on the core functional logic of APIs*. Given the nature of Android API implementation, we argue that path-sensitivity is crucial for accurately summarizing the core functionality and thus important for Replicas detection. Therefore, we model the APIs in the form of *static program paths* and identify the core ones – that is, those implementing the core logic of the API. We then leverage various techniques to further reduce the paths, filtering out other potential noisy, non-core logic in the paths. Finally, we detect Replicas by capturing the similarity of distinct *core paths* across APIs. By performing the comparison exclusively on core paths, our analysis is unaffected by non-core logic. This addresses challenges 1 and 2 since discrepant non-core logic will be disregarded. We refer the reader to [section A.1](#) for a detailed explanation.

---

<sup>1</sup>A semantic block denotes a data independent subgraph, as defined in [24]

# Chapter 4

## OUR APPROACH: RepFinder

To detect and measure the prevalence of Replicas at large scale, we implement RepFinder, an analysis pipeline that combines static program analysis and NLP analysis. [Figure 4.1](#) shows the workflow of RepFinder. The input is a set of Android ROMs and the output is (1) a list of the API pairs <Original, Replica>, and (2) security reports for each pair, summarizing detected access control anomalies if any. The workflow contains four main steps. Next, we explain the details of the individual steps.

### 4.1 Paths Extraction

Given an Android ROM, RepFinder begins by statically analyzing the framework to identify APIs defined in the framework system services. For each identified API, it constructs a context-sensitive call-graph to find *candidate* core paths. We define a candidate core path as the interprocedural control-flow path, composed of a sequence of instructions in the basic blocks from an API’s entry node to a set of target termination nodes – specifically, `return` statements of the calling procedure. RepFinder extracts the paths through

Table 4.1: Paths identified in `removeData()` in `OplusDevicePolicy` – Oppo v.13

Path ID	Instructions	Path Class
1	<code>isOplusReady() return false</code>	Auxiliary
2	<code>isOplusReady() !checkPermission() return false</code>	Failed AC
3	<code>isOplusReady() checkPermission() mCache.get(mUserId) == null Log.d("...") return false</code>	Auxiliary
4	<code>isOplusReady() checkPermission() !(mCache.get(mUserId) == null) ARG<sub>2</sub> == 1 mCache.get(mUserId).mOplusCustomizeData.remove(ARG<sub>1</sub>) updateCustomizeFileHandle return true</code>	Core
5	<code>isOplusReady() checkPermission() !(mCache.get(mUserId) == null) !(ARG<sub>2</sub> == 1) mCache.get(mUserId).mOplusData.remove(ARG<sub>1</sub>) updateFileHandle return true</code>	Core

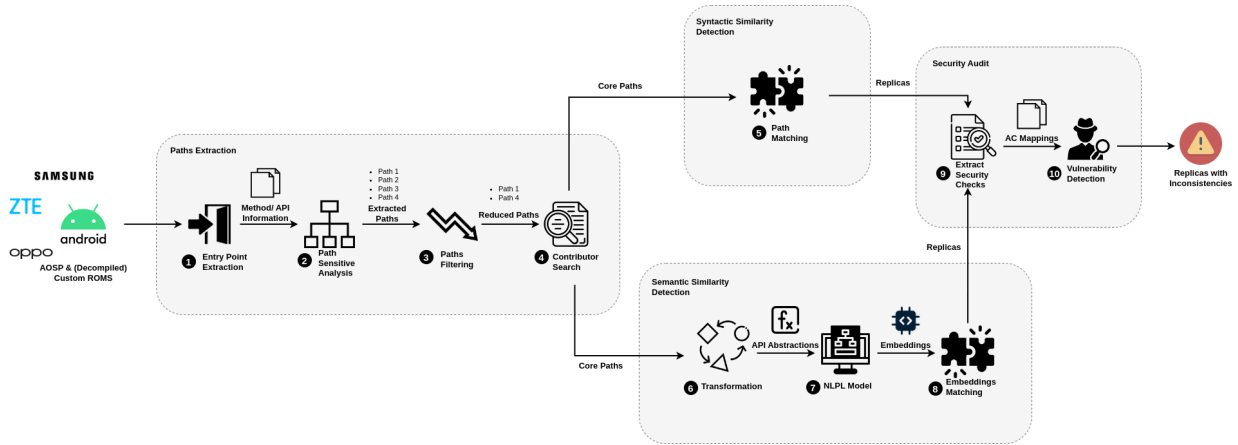


Figure 4.1: Workflow of RepFinder

traversing the graph. Recursive call patterns are avoided during the traversal by checking whether a method is already contained in the currently traversed path.

To address the potential issue of path explosion, we set 3 as a threshold to limit the maximum inter-procedural invocation depth (note that this can be configured). Our analysis reveals that core instructions are often shallow in the call chain and thus analyzing 3 depth levels is sufficient to cover them.

**Example.** Listing 4.1 depicts a code snippet extracted from Oppo API `OplusDevicePolicy.removeData(..)`. For simplicity, we only list the program paths identified by RepFinder at depth 1<sup>1</sup>. The paths are summarized in Table 4.1.

Paths 1 and 2 correspond to the cases where one of the disjunctive conditional checks at line 2 fails. Subsequent code is control dependent on a value maintained in the global map `mUsersCache`. Path 3 reflects the case where the value is null, under which the API returns after logging a debug message (lines 13-15). Paths 4 and 5 depict the core logic functionality. If the second argument equals 1, path 4 (lines 6-7) is executed; otherwise, path 5 (line 9-10) will be.

```

1 public boolean removeData(String name, int datatype) {
2     if (!isOplusReady() || !checkPermission()) return false;
3     Users users = mCache.get(mUserId);
4     if (users != null) {
5         if (datatype == 1) {
6             users.mOplusCustomizeData.remove(name);

```

<sup>1</sup>the number of paths extracted at depth 3 is 7

```

7         updateCustomFileHandle(mCache, mUserId, users);
8     } else {
9         users.mOplusData.remove(name);
10        updateFileHandle(mCache, mUserId, users);
11    }
12    return true;
13 } else {
14    Log.d(TAG, "removeData userId = " + mUserId);
15    return false;

```

Listing 4.1: Oppo API OplusDevicePolicy.removeData(..)

## 4.2 Paths Filtering

Not all program paths in an API correspond to core logic functionality. For example, path 2 in Table 4.1 is rather auxiliary (it enforces access control). We perform a preliminary filtering of *auxiliary* paths by removing those corresponding to failed input validation and failed access control enforcement. We rely on Android domain knowledge and the analysis of AOSP APIs to devise the following static patterns encoding these failures. We note that prior works [65, 56] have demonstrated the promise of using (similar) domain-driven rules for related tasks.

### Rule 1: Failed Input Validation

A path is classified as failed input validation if (1) the last conditional statement, which the path-termination node is control-dependent on, is an input validation check (i.e., where one of the operands is a parameter) and (2) there is no other statement along the path from the check to the termination node except logging statements (e.g., methods in `Log`, `Slog` classes) and their transitively data-dependent instructions. To handle the scenarios where a validation occurs in an invoked method, we propagate input validation information from callee to calling methods. Specifically, if one of the callee method paths is a failed input validation path, we propagate that information to the calling method. If the latter is used in a conditional statement, we classify the failing branch as an input validation.

### Rule 2: Failed Access Control

A path is classified as a failed access control if (1) the last conditional statement on which the path-termination node is control dependent is an access control check (e.g., a permission

check, an app UID check, a user check, etc), and (2) there is no other statement along the path from the check to the termination node except logging statements. We detect and propagate access control information inter-procedurally.

### 4.3 Core Paths Selection

By this stage, RepFinder has reduced the set of candidate program paths. Observe though the paths still include other *auxiliary* paths which cannot be caught by the aforementioned rules (such as paths 1 and 3 in [Table 4.1](#)).

Inspired by prior work [67] aiming to identify *main contributors* in Android app components, we leverage *naming hints* to identify core paths. Specifically, we rely on the observation that Android developers typically follow good naming practices, and thus, core paths will likely include naming clues. For example, the instructions in the core paths 4 and 5 (`mOplusCustomizeData.remove(..)` and `mOpusData.remove(..)`, respectively) bear similarity to the name of the defining API `removeData(..)`. Thus, we can rely on such naming similarity to pinpoint core paths. This hint excludes path 1 from further analysis.

**Instruction de-noising.** Before measuring naming similarity of an API name and enclosed paths instructions, we perform a de-noising step. This ensures exclusion of commonly-used instructions in Android APIs, that are often unrelated to core functionality – consider debug statements and corresponding data-dependent string building instructions. This step is necessary because common instructions not only imply false positives, but also most likely, no <Original, Replica>pair of APIs will look exactly alike.

RepFinder compiles a list of common instructions through frequency analysis. During path extraction, it further conducts data flow analysis to extract corresponding data-dependent instructions. The resulting instructions are then removed from the paths. RepFinder also tackles cases where a noisy instruction exhibits a 1-1 control-dependency on a conditional statement (e.g., `if (DEBUG) Log.d(..)`) by removing the conditional statement from all paths. The de-noising steps also ensures that conditional statements shared with failed paths (e.g., permission checks) are removed from the final paths.

**API and Path Similarity.** We define a function to measure the degree of similarity of a path to its defining API. Formally, given a program path  $P$ , consisting of  $I$  instructions, and an API  $A$  with name  $M$ , a similarity function  $Sim$ , and a threshold  $\theta$ , we consider  $P$  to be a core functional path of  $A$  if  $Sim(P, M) \geq \theta$ . We define  $Sim$  as follows:  $Sim(P, M) = \max(f(i_1, M), \dots, f(i_n, M))$ , where function  $f$  gauges the similarity of an instruction

$i_i$  in the path instruction set  $I$  and the API name. Namely,  $f$  allows assessing whether a given instruction is a *core instruction*. We use the Cosine similarity as the function<sup>2</sup>. To calculate the similarity, we generate and use various information for common instructions. For example, we generate the following instructions (we only show two instruction types for brevity):

- **InvokeInstruction**: we extract invoked method name, defining (super) class, and parameter types. We also resolve constant String parameters (if any) using def-use chains.
- **PutInstruction**: we extract the declared field name, variable type, defining (super) class, and resolve the field value.

Our approach can further estimate core paths using semantic similarity. Specifically, we augment the above instruction information with semantic information as follows: We leverage UniXcoder [38], a unified cross-modal pre-trained NL-PL model, for learning word embeddings. We use the model’s tokenizer to tokenize the API name  $M$  and instruction information pertaining to  $i_i$ , and use the model to generate corresponding embeddings. The cosine similarity function  $f$  calculates the similarity of  $embeddings(M)$  and  $embedding(i_i)$ .

**Core Paths.** At this stage, the analysis outputs a <API, core paths>dictionary per ROM. The dictionary can be reused by the ROM OEM, unless an API is updated. If an OEM specifies the changed APIs, only the corresponding paths will be regenerated, rather than having to completely regenerate the dictionary. This allows OEMs to efficiently perform Replicas detection upon updates.

Next, we discuss how we leverage the generated paths to detect syntactic and semantic clones.

---

<sup>2</sup>Dice similarity function can be used as well.

# Chapter 5

## Similarity detection

We rely on syntactic and semantic similarity to detect Replicas. As mentioned earlier, we use this combination for scalability reasons; although semantic similarity can catch syntactic Replicas, semantic similarity calculation is prohibitively expensive. Thus, we use syntactic similarity for a pair-wise comparison of all APIs in a ROM, and limit semantic similarity calculation to APIs within a service.

### 5.1 Measuring Similarity

In order to quantitatively conclude that two paths are similar, we devise a path similarity function. The function is versatile, in that it can be used to measure both syntactic or semantic similarity of program paths. The function returns a positive value in the range  $[0, 1]$ ; the higher the value, the more similar the paths. Hence, two paths with (syntactic/semantic) similarity value higher than a preset threshold, imply that the defining APIs are Replicas. We refer the reader to [section 8.1](#) for more details on the threshold selection.

Formally, given two APIs  $A_o$  and  $A_r$ , a similarity function  $Sim$  and a threshold  $\theta$ ,  $A_o$  and  $A_r$  are identified as Replicas if there exists a pair of paths  $A_o.P$  and  $A_r.P$  such that  $Sim(A_o.P, A_r.P) \geq \theta$ .

## 5.2 Syntactic Similarity

Under syntactic similarity, paths are represented as a *set of instructions*. Hence, the function  $Sim(A_o.P, A_r.P)$  should capture set-based similarity. While there are many similarity functions, we employ *Jaccard Similarity* because it captures set-based overlaps, unaffected by path length variability. Specifically, given two instruction sets  $P_o$  and  $P_r$ , the Jaccard similarity is computed as:

$$|P_o \cap P_r| / |P_o \cup P_r|$$

that is, the ratio of common set elements in  $P_r$  and  $P_o$  to the size of their union.

Jaccard similarity intuitively penalizes differences between instruction sets, even if one is a proper subset of the other. This property aptly tackles the cases where the target of comparison is an API pair,  $A_o$  and  $A_r$ , such that  $A_o$  invokes  $A_r$  (or vice versa). In such cases, because our path extraction is inter-procedural, the paths  $A_r.P$  will be a proper subset of at least one of the path in  $A_o.P$  (or vice versa). Note that such patterns are common in Android. For example, APIs may offer a parameterized invocation of other APIs (e.g., `PackageManager.installPackage(..)` invokes `PackageManager.installPackageAsUser(..)` with a preset user parameter value). Other APIs commonly invoke utility APIs (e.g., `PackageManager.getPackageUid(..)`). However, the pattern does not necessarily indicate a Replica.

By considering the total cardinality of the sets, the Jaccard similarity can properly identify Replicas as follows. First, the computed similarity will be low when the calling API  $A_o$  invokes other substantial code alongside with API  $A_r$ , implying that the latter is likely a utility method. Second, the similarity will be high when the calling API  $A_o$  invokes no other code except of  $A_r$ , implying that the former serves as a wrapper around  $A_r$ , without providing additional functionality. We refer the reader to [section A.2](#).

While Jaccard similarity is a simple and intuitive function for our task, it may lead to false positives when rare instructions are more informative than frequent instructions. For example, consider the scenario where two paths do not share *core* instructions, but converge on a significant amount of *auxiliary* instructions. Such paths will be wrongly identified as Replicas (i.e., high Jaccard similarity).

The instruction de-noising step ([section 4.3](#)) alleviates the issue through reducing shared and noisy code. We further mitigate the issue by leveraging the notion of *core instruction*, defined in [section 4.3](#). Specifically, we introduce the following filtering property:

**Property 1:** Given two paths  $P_o$  and  $P_r$ , each consisting of  $I$  instructions, if the symmetric difference  $|P_o \Delta P_r|$  contains at least one instruction  $i$  with  $Dice_{sim}(i, O) \geq \beta$ , or  $Dice_{sim}(i,$

Table 5.1: Transformation Rules

IR Instruction	Transformation
SSAAbstractInvokeInstruction	methodName(arg1, arg2, arg3..)
SSAGetInstruction	get(fieldName)
SSAPutInstruction	put(fieldName, val)
SSANewInstruction	init(fieldName)

$R) \geq \beta$ , then APIs  $O$ , and  $R$  are not Replicas. The property essentially implies that if a pair of API paths do not share core instructions, then the APIs cannot be Replicas.

### 5.3 Measuring Semantic Similarity

This task is more challenging as it requires inferring that two paths perform the same behavior even when their code syntax is different. Many deep learning models have been trained to *understand* code. Given a code snippet, the models can generate a vector representation of the snippet which represents the lexicographic, syntactic, and semantic information in the code. While there are many choices of models (e.g., CodeBERT [38], OpenAI’s codex embedding [8]), we use UniXcoder as it has achieved state-of-the-art performance on clone detection [38], and in other related tasks (e.g., code summarization [69] code to NL translation [25], code generation [27]).

Given two paths  $P_o$  and  $P_r$ ,  $Sim$  should capture the similarity between the embedding of  $P_o$  and  $P_r$ . To this end, we employ Cosine Similarity as follows:  $Sim(P_o, P_r) = CosineSim(Enc(P_o), Enc(P_r))$  where  $Enc(P)$  represents the vector embedding of path  $P$ .

**Abstraction.** A program path is in the form of an Intermediate Representation (IR) listing the instructions in the path. Such form is not amenable to UniXcoder, which has been pretrained on code and NL. Therefore, we transform the IR statements to a near-code representation using basic transformation rules presented in Table 5.1.

# Chapter 6

## Security Audit of Replicas

In this section, we evaluate the security of Replicas. Our evaluation investigates two security aspects: (1) access control consistency, and (2) security patch (security logic update) propagation consistency.

### 6.1 Access Control Consistency

Since Replicas implement the same functionality as the original API, they should naturally enforce consistent access control. A weaker or absent access control leading to a replicated functionality will inevitably introduce vulnerabilities. Third-party apps can invoke the weakly-protected Replicas to access the underlying functionality, without satisfying the privilege requirement in the original API.

Extracting access control enforcement along a path is conveniently performed alongside the path extraction procedure ([section 4.1](#)). Observe that this implies that the access control extraction is path-sensitive. During CFG traversal (along a particular path), RepFinder inspects the instructions and identifies those related to access control enforcement. These include (1) instructions invoking methods that enforce Android permissions (e.g., `enforceCallingOrSelfPermission`) and (2) conditional statements where one of the operand in the predicate evaluates to an invocation of known permission checks (e.g., `checkCallingPermission`) and calling app / user property inquiry (e.g., `Binder.getCallingUid`). RepFinder further relies on DefUse chains to extract other access control information such as the permission name, operator, and variables used in the operands. If a path includes multiple access control checks, RepFinder merges them using a logical AND since they all need to be satisfied to reach the core-functionality in the path.

## 6.2 Updates Propagation Consistency

Replicas add a layer of complexity to the already-complicated Android updates procedure [86]. To keep Replicas up-to-date, OEM developers need to carefully track APIs from which the Replicas were cloned and promptly propagate any corresponding updates. Not surprisingly, this task is challenging. Due to the under-regulated nature of custom APIs, OEM developers often lack an explicit knowledge of related APIs (no notion of forks). More importantly, while Replicas provide similar functionality to their original APIs, they often include diverging code. As such, they are difficult to detect without the aid of a Replica detection tool such as ours.

To assess the propagation consistency of updates, , we compare the evolution of access control enforcement in <Original, Replica>pairs, across major versions. More details are discussed in [section 8.3](#).

# Chapter 7

## Large Scale Measurement Study

We conduct our large scale measurement study of Replicas on a corpus of 342 ROMs.

**Implementation and Analysis Configuration.** The static analysis of RepFinder is built on top of WALA[43]. We use WALA’s ZeroXCFA builder to construct context-sensitive call graphs and BasicBlockInContext to make the interprocedural control-flow graph correctly work with context-sensitive call graphs. We use Redis stores for storing embedding and CUDA to work with GPUs.

### 7.1 Dataset Collection

Our dataset spans 7 AOSP ROMs and 335 customized ROMs. We downloaded AOSP ROMs, which are necessary to filter public APIs [37], from the Android official repository [5]. We collected customized ROMs from FirmwareDrive[28], Sammuobile[61], Samfrew [1], and AndroidDumps repository[3]. We wrote crawlers to automatically download the ROMs, when possible.

**Dataset characterization.** Table 7.1 lists the detailed statistics of our custom ROM corpus. As shown, it covers Android’s evolution from version 8 to version 14 (i.e., from SDK 26 to SDK 34). The oldest ROM in our dataset dates back to March 2017, while the newest dates to April 2024 – spanning 7 years.

The custom ROMs cover 10 vendors and 242 distinct device models. According to recent statistics [10], our custom ROM dataset is representative of current Android OEMs. It covers the big players (Samsung, Xiaomi, Oppo, and Vivo), and includes others with

Table 7.1: ROM Dataset

<b>OEM</b>		<b>V.8</b>	<b>V.9</b>	<b>V.10</b>	<b>V.11</b>	<b>V.12</b>	<b>V.13</b>	<b>V.14</b>
<b>Amazon</b>	ROMs	-	2	-	3	-	-	-
	OEM APIs	-	406	-	714	-	-	-
<b>Asus</b>	ROMs	-	-	-	-	-	3	3
	OEM APIs	-	-	-	-	-	834	1572
<b>Lenovo</b>	ROMs	2	10	-	8	7	6	-
	OEM APIs	628	622	-	249	348	1130	-
<b>Motorola</b>	ROMs	-	-	-	-	5	6	3
	OEM APIs	-	-	-	-	812	1307	1708
<b>Oppo</b>	ROMs	-	4	-	19	4	17	3
	OEM APIs	-	336	-	586	304	840	1596
<b>Samsung</b>	ROMs	29	43	11	16	5	28	-
	OEM APIs	3487	2136	2958	2657	2157	1891	-
<b>Tecno</b>	ROMs	-	5	1	4	-	2	-
	OEM APIs	-	88	214	443	-	157	-
<b>Vivo</b>	ROMs	1	1	2	2	3	8	3
	OEM APIs	743	440	806	983	1515	1505	2255
<b>Xiaomi</b>	ROMs	2	5	3	9	28	-	-
	OEM APIs	758	483	457	801	548	-	-
<b>ZTE</b>	ROMs	-	6	7	5	1	-	-
	OEM APIs	-	1012	809	998	499	-	-

Table 7.2: Traces and Embeddings For ROMs

Vendor	Avg Paths / ROM	Min Paths	Max Paths	Total Paths	Total Embeddings
Amazon	20268.4	17004	25132	101342	34013
Asus	9826.0	298	19216	58956	20456
Lenovo	11484.7	9866	18013	321572	105663
Motorola	21169.7	16317	35570	296376	101711
Oppo	14292.5	5974	23710	671752	222101
Samsung	13686.8	13155	32639	3134294	923530
Tecno	11231.2	3076	16319	157237	157237
Vivo	13698.7	383	22838	342468	116653
Xiaomi	12074.3	7880	16797	845202	273133
ZTE	12761	10425	35688	319027	107515

smaller market shares (e.g., Motorola, Lenovo, and Tecno). Samsung and Oppo ROMs are more prevalent in our dataset as there are many repositories dedicated to collecting them. ROMs from other vendors (e.g., Tecno and ZTE) are more difficult to obtain and thus constitute smaller sample sizes in the dataset.

**Preprocessing.** We used a suite of tools for ROM preprocessing. We used a modified version of SALT tool [64] (handles kdz files), `simg2img`[9] (unpacks the super partition), `lpunpack`, `lpmake`[7], and `atcmd`[66] (combine and extract the system partitions). To mount the system partition, we use `imgtool`[6] and extract framework classes using `vdexextractor`[18], `baksmali`[47], `smali`[47], and `apktool`[44].

## 7.2 Analysis Landscape and Complexity.

**Customization Extent.** RepFinder identified all together 44,794 private APIs. Table 7.1, Row #2, details a breakdown of APIs per vendor. As shown, the extent of private APIs varies across vendors where Samsung exhibits the highest number and Tecno introduces the least number. Asus, Lenovo, Motorola, Oppo, and Vivo show a steady increase of APIs in version 12 (version 13 for Asus) to version 14, hence contributing to API sprawls. Other vendors, such as Samsung and Xiaomi, are introducing less APIs in latest versions.

**Path Extraction Statistics.** Table 7.2 reports the summary statistics for the path

extraction phase ( [section 4.1](#)). Column #3 and Column #4 correspond to the minimum and maximum number of paths extracted from a ROM for each vendor. As shown, the paths statistics are generally proportional to the private APIs count; that is, the higher the number of APIs, the higher the number of paths. However, we can see a few exceptions. For example, Amazon introduces less custom APIs compared to Motorola, yet, its min paths count is higher than that of Motorola. We investigated the cases and found that Amazon API implementation is usually more complicated than other vendors (w.r.t. LoC and avg # of conditional statements).

Column #5 in the same table lists the total number of paths analyzed by RepFinder. As Samsung constitutes the biggest player in our ROM corpus, it generates more than 3 million paths. Altogether, RepFinder extracted 6,248,226 paths.

**Semantic Analysis Statistics.** The last column of [Table 7.2](#) reports the total embeddings generated for all paths. On average, RepFinder generates 6,155 embeddings, per ROM – totalling 2,062,012 for the entire corpus. Given the popularity of Samsung ROMs in the dataset, its corresponding paths generate the largest number of embeddings. Observe that the embedding count (Column #6) in [Table 7.2](#) is less than the path count (Column #5) because the semantic similarity detection ([section 5.3](#)) is only concerned with *core functional paths*, which amount to two paths on average, per API.

# Chapter 8

## Replicas Outlook

In this section, we answer the following research questions:

- RQ1: How pervasive are Replicas in the Android ecosystem?
- RQ2: Are Replicas inherited or newly introduced across major versions?
- RQ3: How is the security landscape of Replicas?

### 8.1 RQ1. Replicas Prevalence

Our study shows that Replicas are prevalent in the Android fragmented ecosystem. Among the 44,794 custom APIs, RepFinder discovered 4339 (9.7%) instances of Replicas. [Figure 8.1](#) depicts a breakdown of Replicas per OEM. Each individual ROM contains Replicas. The pervasiveness varies across vendors and versions. On average, the ratio of Replicas ranges from 9.3% in Xiaomi to 15.6% in Amazon (for all available versions combined, per vendor). It reaches a max of 17.2% in Lenovo (version 11). Interestingly, earlier versions of Lenovo (versions 8 and 9) records the least number of Replicas (<5%). The evolution trend of Replicas varies across vendors. Amazon, Motorola, and ZTE exhibit the most consistent trend – the ratio of Replicas increases steadily across versions. For example, the ratio of Replicas in ZTE increases from 4.2% to 16% in version 8 to 12. Other vendors show a random pattern. For example, the ratio of Samsung’s Replicas decreases from versions 9 to 12, but slightly increases again in version 13. Similarly, Oppo’s Replicas decrease from version 9 to 13 but increase in version 14. Tecno and Xiaomi follow a more random pattern.

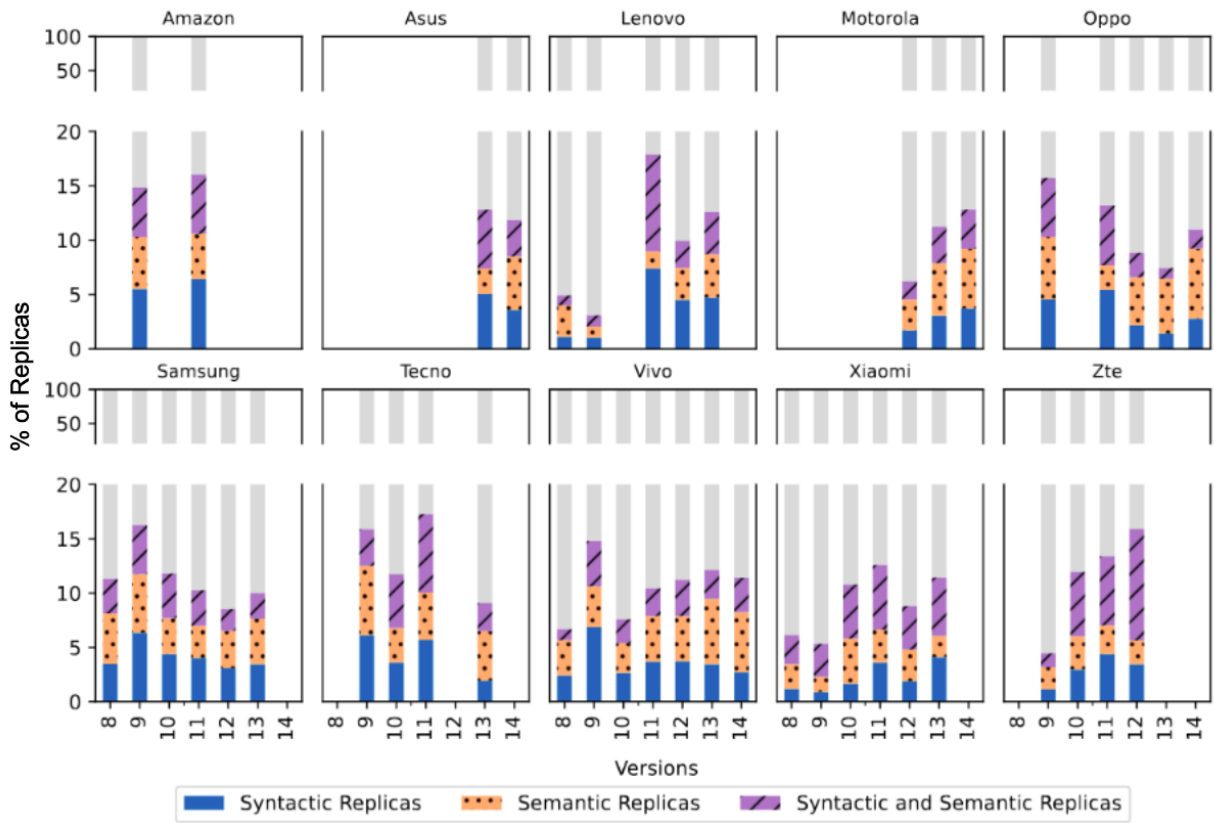


Figure 8.1: Replicas Breakdown

## Syntactic vs Semantic Replicas

Figure 8.1 further depicts a breakdown of semantic and syntactic Replicas. As mentioned earlier, RepFinder performs syntactic detection across all APIs in a ROM and limits semantic detection to in-service APIs. The results can thus be different. The Replicas, classified as both semantic and syntactic correspond to in-service API pairs that are syntactically similar. On average, they constitute 36.6% of all Replicas. Syntactic-only Replicas reflect similar APIs across system services (34.4% of the all Replicas), while semantic-only Replicas correspond to in-service API pairs that are semantically similar but syntactically different (29% of all Replicas).

**RQ1 Findings:** Replicas are prevalent in the Android fragmented ecosystem. On average, the ratio of Replicas ranges from 9.3% in Xiaomi to 15.6% in Amazon (across the analyzed versions) .

## RepFinder Accuracy

Due to the lack of ground truth, we resort to manual validation to estimate accuracy. We randomly select 3 ROMs and sample 339 Syntactic and 321 Semantic Replicas. Two authors investigated the decompiled implementation of <Original, Replica>pair sample. A case is considered a false Replica if the authors agree. Out of the 339 Syntactic cases, 88% are identified as true Replicas, while out of the 321 Semantic cases, 73% are considered true instances – implying a 12% and 27% FP rates for Syntactic and Semantic analyses, respectively. The FPs are mainly due to the following: (1) in a few cases, auxiliary paths were classified as core because of high naming similarity, and (2) the semantic measure over-approximates similarity in paths instructions (e.g., `LocationManager.getLocationInfo(..)`, which returns location metadata is considered similar to `LocationManager.getLocation(..)`, which returns actual location coordinates).

## Sensitivity to Similarity Threshold

We examine the sensitivity of RepFinder to variations in the similarity threshold used to detect Replicas (chapter 5). We run the detection for 3 representative ROMs, under multiple threshold values, varying from 0.65 to 0.9. Due to the lack of ground truth, we resort to manual validation to assess the results. Specifically, to estimate the FP

Table 8.1: Impact of Similarity Threshold

Threshold	Syntactic Similarity		Semantic Similarity	
	FP	FN	FP	FN
0.65	0.27	0	0.53	0
0.7	0.24	0	0.46	0
0.75	0.2	0.02	0.45	0
0.8	0.17	0.03	0.38	0.02
0.85	0.12	0.05	0.27	0.04
0.9	0.11	0.09	0.24	0.14

rate under each threshold, we randomly sample 100 reported cases (50 syntactic and 50 semantic) and manually check if they correspond to True Replicas. To gauge potential missed True Replicas (FNs), we construct a sample of 100 manually detected Replicas. We then check if there are reported by RepFinder. As shown in Table 8.1, the threshold 0.85 achieves the best trade-off, leading to lower FPs and fewer missed true Replicas.

## 8.2 RQ2. Addition, Removal, and Inheritance Trends

As depicted in Figure 8.1, with the exception of ZTE and Motorola, the average ratio of Replicas is lower in version 13+ than in earlier versions. Although this is generally a good indicator that OEMs remove Replicas, we found that they also *introduce new instances*.

To showcase this trend, we report (1) new Replicas introduced in each version (i.e., for a given version  $X$ , we count the number of Replicas that did not appear in version  $X - 1$ ), and (2) removed Replicas in each version (i.e., for a given version  $X$ , we count the number of Replicas that appeared in version  $X - 1$  but not in  $X$ ). Figure 8.2 reports the results. As shown, 4 out of the 7 vendors (with an available version 12 and 13 ROM in our dataset) add more Replicas in version 13 than in version 12. The removal pattern is mostly consistent across versions 12 and 13 (except Samsung and Motorola). We can also see that the ratio of added Replicas is higher than that of removed ones.

**RQ2 Findings:** Although inherited Replicas are commonly removed by vendors in later versions, new Replicas are also introduced consistently in newer versions. This finding clearly demonstrates that Replicas are not an issue of the past.

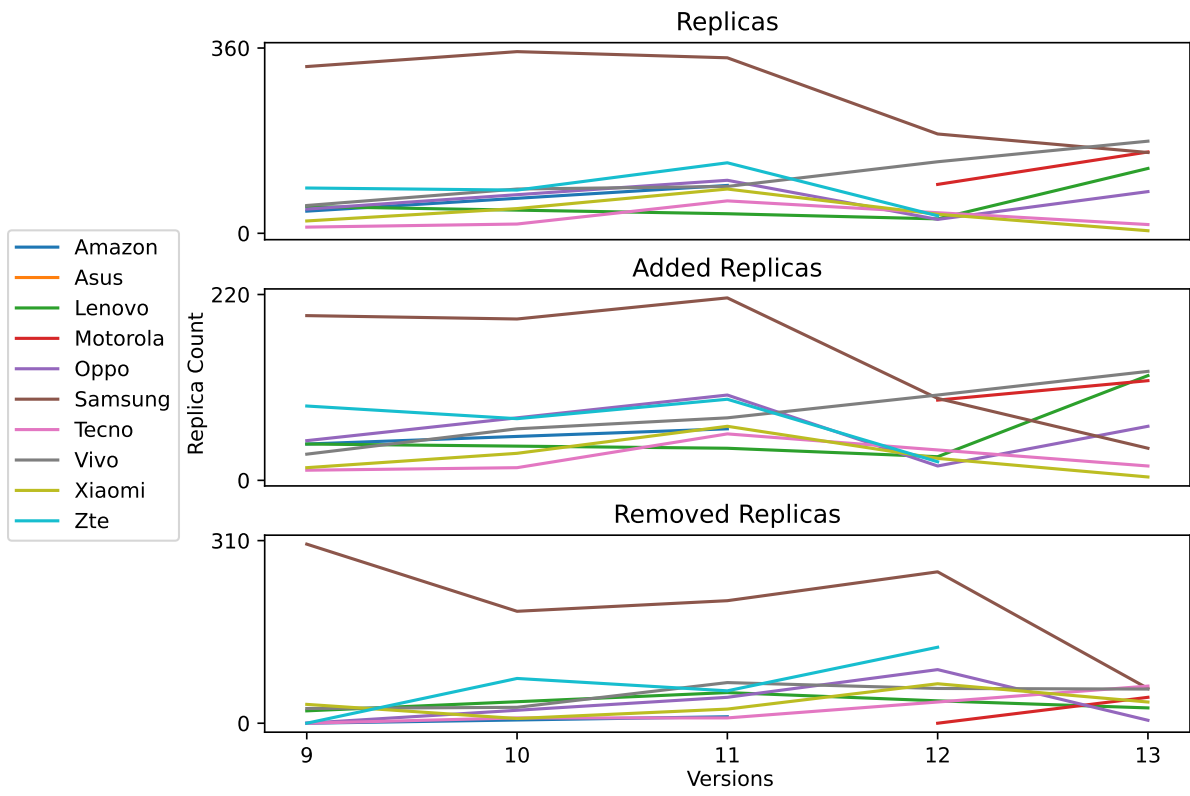


Figure 8.2: Addition, Removal Trends, per Vendor

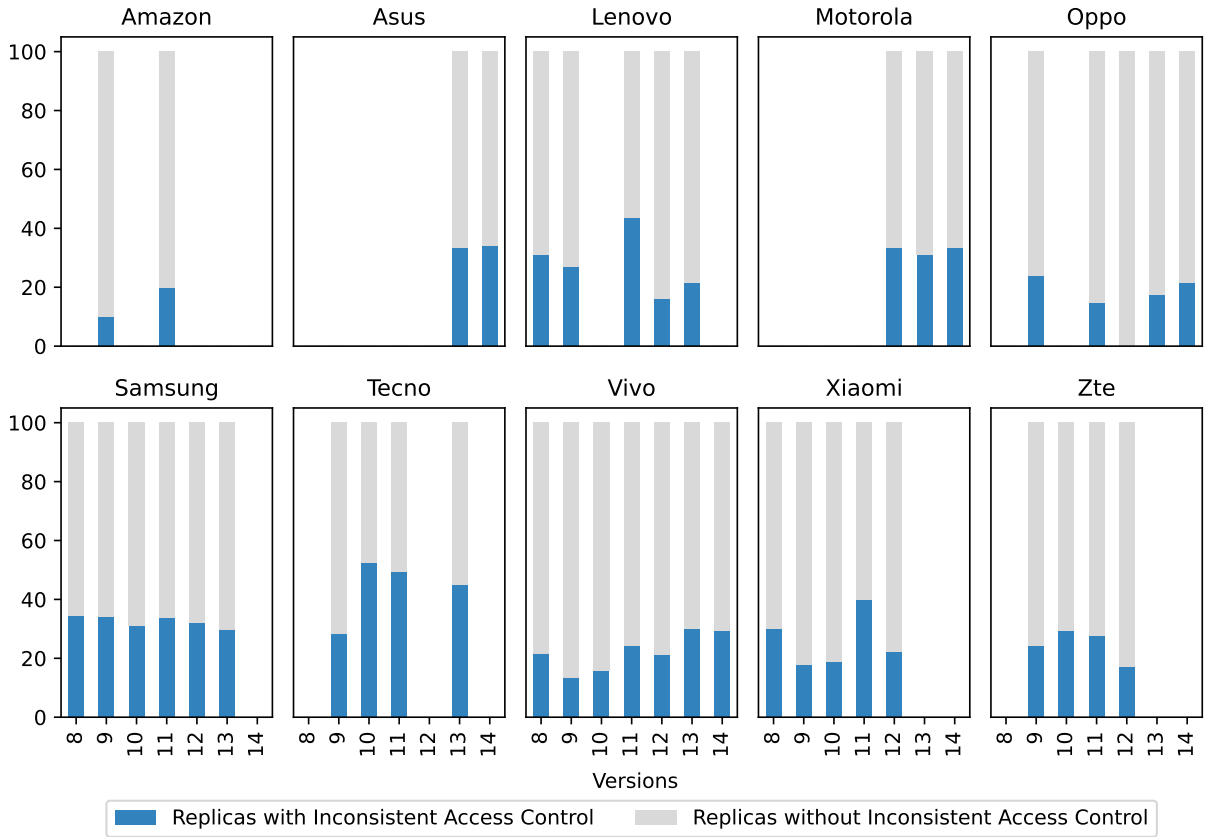


Figure 8.3: Access Control Inconsistencies in Replicas

### 8.3 RQ3. Replicas Security Audit Results

#### Inconsistent Access Control

Figure 8.3 reports the ratio of Replicas with access control inconsistencies, per OEM and across versions. As shown, RepFinder identified 1642 likely inconsistent security checks among the discovered Replicas. Note that the flaws correspond to cases where the Replica implements a weaker access control check than the original API, or misses to enforce it all together. On average, 37% of Replicas across all versions were found to include an access control inconsistency. The flaws are common to all OEMs, reaching more than 40% under-protected Replicas in Tecno, Lenovo, and Xiaomi. By comparison, Amazon exhibits significantly fewer flaws (16% on average) and the lowest ratios (10%) of under-protected

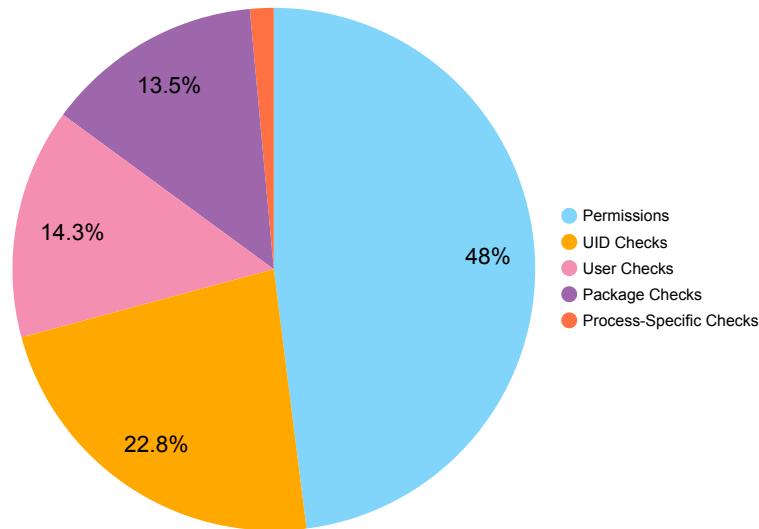


Figure 8.4: Distribution of Access Control Checks

Replicas.

### Distribution of Access Control Inconsistencies

RepFinder uncovers inconsistencies caused by various misuses of security features. A breakdown is depicted in [Figure 8.4](#). Inconsistent permission checks are the most prevalent in Replicas with an average ratio of 48%. Examples include `getStreamVolumeForSlientKey` missing `QUERY_AUDIO_STATE` permission in Vivo iQOO (v.11) and `setEarlyWakeUp` is missing `DEVICE_POWER` permission in Samsung Galaxy S10 (v.9). Missing UID checks are the second cause of inconsistencies with an average ratio of 22.8% among under-protected Replicas. Other security features (e.g., user-specific, process-specific and package-related checks) are less pervasive since there are less commonly used in APIs compared to permission and UID checks.

### Security Updates Consistency

Here we devise the following experiment: For each  $\langle \text{Original}, \text{Replica} \rangle$  API pair, we compare their access control implementation across subsequent versions (if both appear in two subsequent versions). If the Original API’s security implementation changes but the

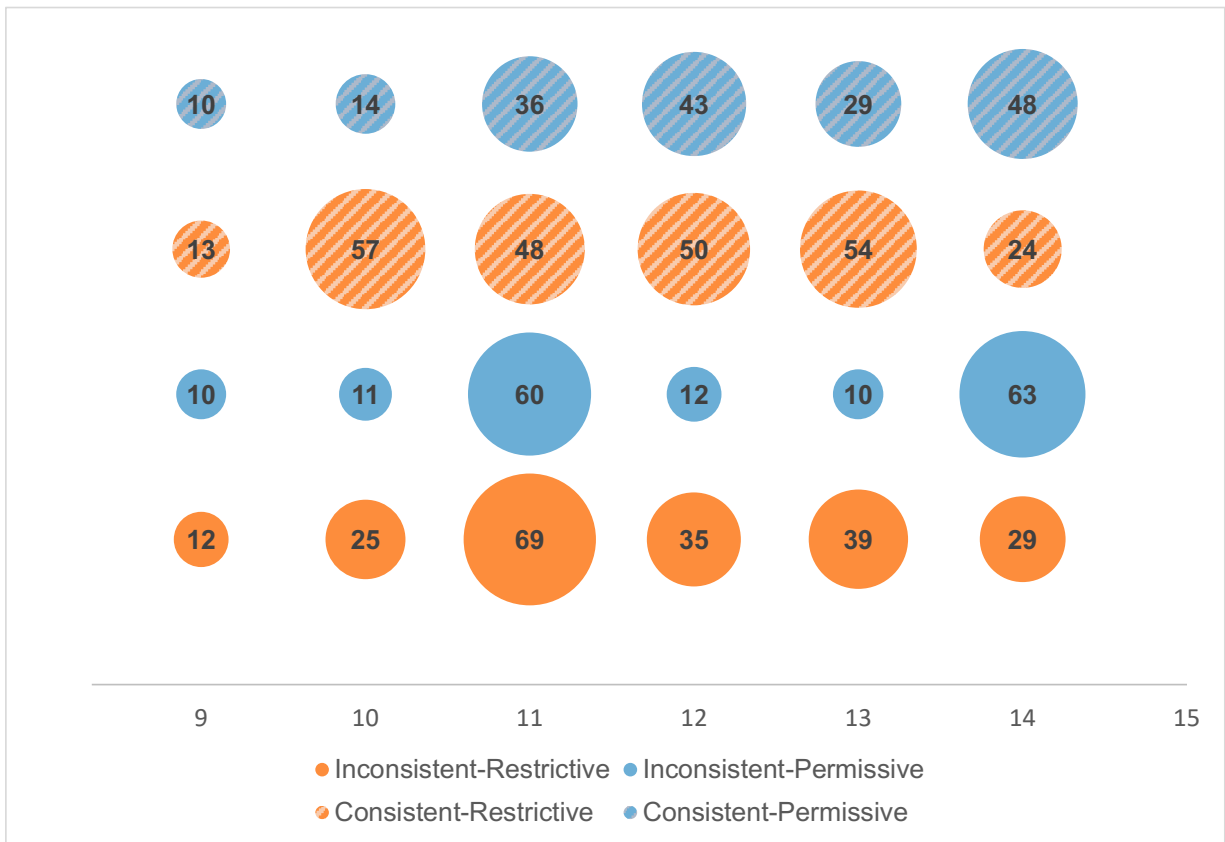


Figure 8.5: Security Updates Consistency in Replicas

Table 8.2: Vulnerabilities Manually Verified

Vendor	Model	Location	Impact	Status
ZTE	Axon 40 Ultra	InputManagerService	KeyLogger	Ack
Vendor A	Device X	PackageManager	Read Any Application directory	Ack
Vendor A	Device X	BackupManager	Write to Any Application directory	Ack
Tecno	Spark	TelephonyManager	Write to Secure System Settings	Ack
Tecno	Spark	TelephonyManager	Read Build Serial Id	Ack
Vivo	Vivo X60	VivoPhoneLockService	Read Sim Status	Reported
Vivo	Vivo X80	AudioService	Read Volume State	Reported

Replica’s does not, we inspect and categorize the change as follows: (1) Consistent – that is the change is performed in both APIs (both Original and Replica APIs are upgraded similarly), (2) Inconsistent – only the Original API is upgraded. We further categorize the transition based on the nature of the change – (1) Restrictive: the update is an addition of an access control check, (2) Permissive: the update removes an access control.

Figure 8.5 shows the results aggregated for all vendors across versions. The categories *Consistent-Restrictive* and *Consistent-Permissive* indicate that a security update was carried out successfully in Replicas. Fortunately, such cases are common. However, the categories *Inconsistent-Restrictive* and *Inconsistent-Permissive*, which are more alarming, imply that a security upgrade was not properly propagated in the <Original, Replica>pairs. In particular, we observe that 69 (original) APIs were upgraded to include a new access control check in version 11; however, their corresponding Replicas did not. Such alarming cases cover 209 Replicas across all versions. The lapse clearly warrants a closer look.

## Exploiting Replicas

To showcase the security impact of the under-protected Replicas, we select a few of instances and build proof-of-concept (PoC) exploits. Our selection is driven by device availability and by the severity of the missing access control (e.g., a missing `INSTALL_PACKAGE` permission is easily exploitable compared to a missing user check). Table 8.2 lists the exploited cases. As shown, we were able to exploit 5 high-impact Replicas, leading to security-critical consequences such as spying on user taps (on ZTE), and even breaking the Android app isolation mechanism. Particularly, two Replicas allow a third-party app to read a random app’s directory and even to write content to it without any permission.

**RQ3 Findings:** On average, 37% of Replicas were found to include an inconsistency. Besides, security patches were not properly propagated in 375 <Original, Replica>pairs in version upgrades. This finding points to *unnecessary* security hazards in Replicas and the urgent need to debloat them.

# Chapter 9

## Comparison With other Tools

Android custom APIs are in decompiled Java bytecode format. Hence, we initially limited our comparison to tools that can handle Java bytecode clone detection. We used LibScan [76], a state-of-art clone detection in Android apps. However, it was not suited for our task as it cannot detect smaller-size clones such as Android APIs<sup>1</sup>.

We then extended our comparison to other open source tools that can handle Java clone detection. We obtained access to SourcererCC [59] and CCAAligner[72], two state-of-the-art token-based code clone detectors. To allow comparison, we used JADX [63], to produce (approximate) Java source code of APIs from Android Dex file. We used the default threshold values specified in SourcererCC and CCAAligner papers to detect API clones.

Due to the manual efforts required for conversion, we construct a smaller corpus that includes (1) the latest AOSP framework source codebase (version 14), and (2) 6 randomly sampled custom ROMs.

**Comparison.** We used methods defined in system services classes as the input for SourcererCC and CCAAligner. In total, SourcererCC and CCAAligner detected 6141 and 5,154,989 cloned methods, respectively. Since not all methods in the system services are APIs (i.e., some are internal methods), we filter the output to include only cloned APIs, resulting in 106, and 5274 API clones, respectively.

Table 9.1 reports the detection results. Observe that we only compare against the syntactic Replicas of RepFinder for fairness, as the tools are limited to finding syntactic clones.

---

<sup>1</sup>as confirmed by the authors

Table 9.1: Clones Detected by Tools

Vendor	Version	ROM	RepFinder-Syn	SourcererCC	CCAligner
Amazon	11	Fire HD 8	64	0	599
Asus	14	ROG Phone 7	85	7	973
Lenovo	13	P11 Pro Plus	76	8	840
Motorola	14	Moto g84	85	11	581
Oppo	14	Find X7 Ultra	48	0	259
ZTE	11	Blade V30 Vita	68	4	661

**Findings.** Column #5 reports the detected syntactic Replicas reported by RepFinder. We similarly follow our aforementioned manual investigation to confirm FPs (11%). As shown, RepFinder identifies more Replicas than SourcererCC. We manually confirm (for 30 randomly selected cases) that SourcererCC misses them because of the prevalence of non-core logic.

On the contrary, CCAligner reports significantly more clones than RepFinder. We manually analyze 5% of the clones and found that the majority are false positives. The primary reason is that CCAligner aims to find *large-gap clones* - that is, clones that reuse code with many additions/subtractions of code lines. A significant portion of Android APIs (particularly those within the same service), follow this pattern but they are not necessarily Replicas.

This demonstrates that token-based code clone detection is not suitable for finding Replicas.

# Chapter 10

## Threats to Validity

### 10.1 Internal Threats to Validity

The primary threat to the validity concerns the accuracy of the abstracted code used by UniXcoder to perform semantic comparison and clone detection. Since Android custom APIs are in decompiled Java bytecode, we devised a few abstraction rules to estimate a near-source code representation. Although this decision is our best effort, it cannot guarantee an accurate representation as expected by the model.

Besides, our tool can tackle the task of finding semantic similarity, by extracting, pin-pointing core program paths and limiting clone detection to these paths. UniXcoder leads to a relatively high false positive rate. We note though that our proposed semantic similarity calculation is highly versatile, in that it can be easily replaced with other models. For example, newer and more powerful models (such as LLMs) can be used instead to alleviate the current false positive rate. We plan to address this in future work.

### 10.2 External Threats to Validity

Unlike the application layer, the use of obfuscation is uncommon at the Android framework layer. In the 342 ROMs we analyzed, we did not spot obfuscated APIs. If the practice becomes more prevalent among OEM framework developers, our tool will be inherently limited.

# Chapter 11

## Discussion

This study reveals the widespread presence of code clones, referred to as Replicas, in Android customizations. Through a comprehensive analysis of over 300 ROMs, we identified an average of 141 Replicas per ROM, indicating that vendor developers frequently introduce APIs without verifying whether a similar Replica already exists. However, we also observed that some vendors are implementing measures to reduce the occurrence of these Replicas. Our evolutionary study across different Android versions shows a trend of decreasing Replicas in newer versions, particularly in version 13.

Replicas not only contribute to the bloat of the Android framework but also present significant security concerns. On average, 37% of Replicas exhibit inconsistent access control enforcement compared to their AOSP counterparts. This inconsistency rises to 51% for some vendors, highlighting a concerning lack of effective security mechanisms in the APIs introduced by developers in customized Android versions. Our experiments further demonstrate that security enforcement in Replicas is not consistently updated in line with corresponding AOSP APIs, underscoring the need for stricter regulation of Replicas within the framework.

In this study, we also exploit under-protected Replicas to showcase the associated risks, emphasizing the importance of addressing these security gaps. Our developed tool, RepFinder, plays a crucial role in identifying the prevalence of Replicas within customized frameworks and highlights the need for more stringent regulation of framework customizations. By using this tool, OEM developers can identify potential Replicas when introducing new APIs, leading to improved security enforcement and reducing unnecessary framework bloat.

# Chapter 12

## Related Works

In this section we discuss the related works done in Android framework analysis, OEM customization hazards, and code clone detection.

### 12.1 Android Framework Analysis

The security of the Android framework has been the subject of extensive analysis, particularly in the context of identifying permission maps and access control inconsistencies. A significant challenge in this area is the lack of standardized regulations and policies within the Android community regarding permission specifications. Consequently, numerous efforts have been dedicated to mapping permissions to detect and prevent issues related to over-privilege and under-privilege. In this section, we will explore the research that has identified security inconsistencies within the Android framework.

Stowaway[32], is one of the earlier works that uses dynamic analysis to generate the maximum set of permissions needed for an application and compares them to the set of permissions actually requested. To overcome the limitations of dynamic analysis, PScout[19] uses static analysis to extract and analyze Android's permission specifications across different versions of the OS. By analyzing 4 different versions of Android it finds that 22% of non-system permissions could be hidden if only documented APIs are used. Backes et al. [21] work on improving the static analysis of the Android application framework. The authors highlight the challenges in analyzing the framework due to its complex design patterns, which differs from those in the application layer. To address this, they develop a tool called Explorer, which systematically analyzes the framework's control and data flows, improving the precision and efficiency of security analysis from previous works.

To further refine the analysis of the framework, Aafer et al.[12] introduce Arcade. The tool deals with imprecise protection mapping from previous approaches. The authors introduce path-sensitive analysis and a novel graph extraction technique that leads to a more precise protection mapping. To improve the analysis of the framework, Dawoud et al.[26] build Dynamo, a dynamic analysis tool that uses grey-box fuzzing. Their tool complements the other static analysis tools by adding more coverage.

Closely related to our security audit are Kratos[62], AceDroid[11], and ACMiner[45] which all attempt to identify flawed access control in APIs by performing inconsistency analysis across APIs converging on the same functionality. Kratos [62] uses precise call graph for to perform inconsistency analysis. Through this, they uncover third-party applications with insufficient privilege accessing privileged resources. AceDroid[11] addresses a limitation that previous approaches did not handle, the complexity of access control checks and they propose a normalization technique for access control checks. IAceFinder[88] and FReD [46] take access control inconsistency analysis one step further by relying on security oracles across various layers of the Android software stack.

More recently, tools have emerged that use probabilistic inference to analyze the Android framework. Poirot [30] applies probabilistic techniques to recommend security protections for the framework, while Bluebird [67] audits Android APIs for potential security gaps by analyzing preloaded apps and extracting security indicators from their app-side logic.

Our work contributes to this body of literature by studying a different phenomena, Replicated code at the framework layer, which can introduce access control flaws. We also use static analysis techniques used in previous research works to model the security of each Replica before detecting inconsistencies.

## 12.2 OEM-Customization Hazards

The security risks introduced during Android OEM customization have been extensively studied across different layers. While vendors customize the framework to add features and functionalities, this process is often highly unregulated, leading to the introduction of security vulnerabilities.

At the application layer, the work [75] analyzed pre-installed apps in 10 custom ROMs and identified various issues such over-privilege and permission re-delegation. Aafer et al. [14] systematically identified security features that, if altered during the customization, can introduce potential risks. They conducted a large-scale differential analysis on

hundreds of custom images and detected risky security configuration such as permissions holding different protection levels.

At the framework layer, prior research analyzed private APIs and reported other customization hazards. Invetter[84] reported the high prevalence of invalid input validations in private APIs. The authors use a combination of machine learning and static analysis to locate sensitive problematic input validations. El-Rewini et al. [29] uncovered that some private APIs are Residuals, i.e., no longer used in a ROM, but still defined. They detect residuals by analyzing both the application and framework layers. Similar to Replicas, Residuals were found to include security flaws.

At other layers, ADDICTED [90] finds security hazards introduced during the customization of device drivers. SEPAL [79] automatically examines customized SEAndroid policy rules using NLP technique and an ML model to predict unregulated rules. More broadly, Possemato et al. [57] perform a longitudinal study to check the compliance of customized devices with various Google’s regulations imposed on Android-branded devices. The study identified significant violations of the guidelines. Furthermore, Hay et al. [39] discovered bootloader vulnerabilities in vendor customizations. Through a combination of static and dynamic analysis, Zhang et al.[81] investigate the effects of customization on the ION unified memory management interface in ARM-based Android devices. They discover vulnerabilities such as memory dumping and denial-of-service attacks that arise due to customization of ION.

Our work identifies another hazard of OEM customizations. We discover that APIs added during the customization process are often Replicas that are under-protected than the original AOSP API.

## 12.3 Code Clone Detection

Clone detection techniques have been developed to serve various purposes. In the Android community, clone detection techniques have been widely adopted to detect clones of applications and malware within applications. Code clone detection is a highly studied area and there have been many different approaches, including token-based, neural networks, and machine learning.

Many tools have been developed to detect malware and vulnerable third-party libraries in applications. A closely related work is by Fisher et al.[33] They detect harmful code being copied from StackOverflow into Android Apps, by first filtering security-related StackOverflow code snippets using security API calls and library calls. They then detect these

snippets in Android Apps using program dependency graphs. Their similarity detection is similar to our in the sense that they use Jaccard similarity however, they try to match similar code snippets in Android Apps. RepFinder matches APIs depending on their core paths extracted from interprocedural control flow graphs.

DroidClone [17] detects malware clones in Android apps through Malware Analysis and Intermediate Language and matches clones based on the signatures of the bytecode blocks. DroidClone effectively reduces the impact of obfuscation and detects malware variants that are syntactically different but semantically similar. Testing with four mobile security tools showed detection rates ranging from 20.2% to 79.6%.

To efficiently and accurately detect third-party libraries (TPLs) in Android apps, LibScan [76] uses similarity-based techniques, even when advanced obfuscation techniques like repackaging and control-flow randomization are used. ATVHunter [80] is another tool that can precisely determine the existence of vulnerable third-party libraries in applications. The authors first create a database of potential TPLs. The authors then extract control flow graphs from within applications to match them with the entries in the database. By checking opcodes from each basic block, they also determine the exact TPL version. LibPecker [85], compared to other tools, is highly resilient to obfuscation. It creates a signature for each library and tries to detect it within applications. To cater to customized code within the applications, the authors use fuzzy class matching, adaptive class similarity threshold, and weighted class similarity score when calculating library similarity. There are many other tools that use similarity-based techniques to detect TPLs in applications [83, 20].

These methods are effective because libraries and malware are often integrated into applications without modification, making them easily detectable. However, these methods cannot be leveraged for our task, since opcodes and graphs of two non-Replica APIs can be highly similar – thus leading to FPs. Our solution addresses this by relying on core-logic program paths.

There are also many instances of cloned applications within the Android marketplace. Many tools have been developed to detect cloned applications. DNADroid [24] leverages program dependency graphs in methods of candidate apps to detect cloned Android apps. The authors collected 75000 applications from 13 different app markets and ran DNADroid. They found 141 applications that have been cloned.

Zhou et al. [89] present a fast, scalable approach to detect piggybacked apps by decoupling an app’s code into primary and non-primary modules and using feature fingerprinting to identify piggybacked apps based on shared primary modules. The proposed method was tested on 84,767 apps from various Android markets, processing them in less than nine

hours and revealing that 0.97% to 2.7% of the apps were piggybacked, primarily for ad revenue theft and malicious payloads. DAPASA [31] is another tool to detect piggybacked apps through sensitive subgraph analysis. By analyzing the different invocation patterns of sensitive APIs in the malicious and legitimate parts of an app, DAPASA generates a sensitive subgraph (SSG) that highlights suspicious behaviors. These patterns are then quantified into five features, which are used in machine learning models to detect whether an app is piggybacked or benign.

Numerous token-based approaches have been proposed to detect clones in source code (in Android Java code or other languages). SourcererCC [59] uses an inverted index to efficiently filter out candidate blocks for a piece of code, whereas CCaligner [71] uses a sliding window to match code using a specific edit distance. CCLearner [53] uses extracted tokens to train a classifier to detect clones. Gode et al.[35] use incremental analysis for on-the-fly detection and evolutionary clone analysis. Our dataset contains decompiled Java bytecode, therefore token-based approaches in source code cannot be efficiently applied as demonstrated in the Evaluation section.

Structural-based approaches have also been employed in code clone detection. CCSsharp[70] is a PDG-based (Program Dependency Graph) clone detector that enhances efficiency by utilizing structure modification and characteristic vector filtering to significantly reduce computational overhead. To further optimize the computation time in PDG-based clone detection, Zou et al.[92] introduced CCGraph, a tool that leverages graph kernels for more efficient processing. Meanwhile, for better scalability, Decker[48] is based on the characterization of subtrees with numerical vectors in the Euclidean space and leverages Euclidean distance metric to match them. Hu et al.[41] introduce an efficient AST-based clone detector called TreeCen. This tool treats the abstract syntax tree (AST) as a social network and applies centrality analysis to each node, preserving the detailed structure of the tree whereas, Wu et al.[77] convert the complex structure of an AST into simpler Markov chains and measure the distances between all states within these chains to detect clones. Baxter et al.[22] also leverage ASTs to find clones, they use a well known compiler method for detecting common sub expressions.

Various ML-based tools have been proposed to detect clones. For example, SJBCD [68] uses a Siamese Neural Network to detect Java code clones. Wei et al.[74] approach clone detection as a supervised learning-to-hash problem and introduce an end-to-end deep feature learning framework for functional clone detection. Natural language (NL) processing models that excel in both generation and understanding tasks also demonstrate high accuracy in detecting code clones. Notable among these are models like [15, 73, 54], which are trained on NL and PL corpus. UniXcoder [38] enhances this approach by using cross-modal contents to create a unified pre-trained model that can be used to effectively detect

code clones. We use UniXcoder’s NL-PL model to generate semantic embeddings of core paths.

# Chapter 13

## Conclusion

We perform the first large-scale study across 342 ROMs to uncover Replicas. To this end, we develop RepFinder, a clone detection tool that identifies syntactic and semantic Replicas by leveraging core-logic program paths. Our study reveals the widespread presence of Replicas in the Android ecosystem. More importantly, it highlights that Replicas not only contribute to bloated Android codebases, but also present significant security concerns. Our study thus emphasizes the importance of debloating Replicas and addressing the underlying security gaps. RepFinder can be of great benefit to OEM developers for automatically detecting Replicas during Android customization.

# References

- [1] Original samsung firmware updates, 2019.
- [2] Android api documentation. <https://developer.android.com/reference>, 2024.
- [3] Android dumps. <https://dumps.tadiphone.dev/dumps/>, 2024.
- [4] Android for developers. <https://developer.android.com/>, 2024.
- [5] Android images. <https://developers.google.com/android/ota>, 2024.
- [6] Imgtool. <https://newandroidbook.com/tools/imjtool.html>, 2024.
- [7] Lpunpack and lpmake, 2024.
- [8] Openai codex, 2024.
- [9] simg2img. <https://formulae.brew.sh/formula/simg2img>, 2024.
- [10] Statcounter global stats 2024, 2024.
- [11] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.
- [12] 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.

- [13] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, page 1248–1259, New York, NY, USA, 2015. Association for Computing Machinery.
- [14] 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, Austin, TX, August 2016. USENIX Association.
- [15] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.
- [16] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE Access*, 7:86121–86144, 2019.
- [17] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. Droidclone: Detecting android malware variants by exposing code clones. In *2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 79–84. IEEE, 2016.
- [18] Anestisb. vdextractor: Tool to decompile extract android dex bytecode from vdex files, 2024.
- [19] 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, CCS '12*, page 217–228, New York, NY, USA, 2012. Association for Computing Machinery.
- [20] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 356–367, New York, NY, USA, 2016. Association for Computing Machinery.

- [21] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Ocateau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *Proceedings of the 25th USENIX Conference on Security Symposium, SEC'16*, page 1101–1118, USA, 2016. USENIX Association.
- [22] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant’Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377. IEEE Computer Society, 1998.
- [23] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33, 2007.
- [24] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*, pages 37–54. Springer, 2012.
- [25] Anh T. V. Dau, Jin L. C. Guo, and Nghi D. Q. Bui. Docchecker: Bootstrapping code large language model for detecting and resolving code-comment inconsistencies, 2024.
- [26] Abdallah Dawoud and Sven Bugiel. Bringing balance to the force: Dynamic analysis of the android application framework. 01 2021.
- [27] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. Traced: Execution-aware pre-training for source code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*, New York, NY, USA, 2024. Association for Computing Machinery.
- [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, CCS '21*, page 1598–1611, New York, NY, USA, 2021. Association for Computing Machinery.
- [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] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. Dapasa: Detecting android piggybacked apps through sensitive subgraph analysis. *Trans. Info. For. Sec.*, 12(8):1772–1785, aug 2017.
- [32] 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, CCS '11*, page 627–638, New York, NY, USA, 2011. Association for Computing Machinery.
- [33] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy-paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, 2017.
- [34] Roberto Gallo, Patricia Hongo, Ricardo Dahab, Luiz C. Navarro, Henrique Kawakami, Kaio Galvão, Glauber Junqueira, and Luander Ribeiro. Security and system architecture: comparison of android customizations. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [35] Nils Göde and Rainer Koschke. Incremental clone detection. *2009 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, 2009.
- [36] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. Multi-threshold token-based code clone detection. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 496–500, 2021.
- [37] Google. Google Developers platform architecture, 2022.
- [38] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unix-coder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.
- [39] Roei Hay. fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.
- [40] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. BigMAC: Fine-Grained policy analysis of android firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 271–287. USENIX Association, August 2020.

- [41] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. Treecen: Building tree graph for scalable semantic code clone detection. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.
- [42] Yu-Liang Hung and Shingo Takada. Cppcd: A token-based approach to detecting potential clones. *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 26–32, 2020.
- [43] IBM. Wala: T.j. watson libraries for analysis.
- [44] iBotPeaches. Apktool: A tool for reverse engineering android apk files, 2024.
- [45] Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android’s middleware. *CoRR*, abs/1901.03603, 2019.
- [46] 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, Boston, MA, August 2022. USENIX Association.
- [47] JesusFreke. Smali: Smali/baksmali, 2024.
- [48] Lingxiao Jiang, Ghassan Mishherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering, ICSE ’07*, page 96–105, USA, 2007. IEEE Computer Society.
- [49] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [50] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–163, 2018.
- [51] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264, 2018.

- [52] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 249–260. IEEE, 2017.
- [53] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara G. Ryder. Cclearner: A deep learning-based clone detection approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, 2017.
- [54] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.
- [55] Microsoft. Codebert - unixcoder. <https://github.com/microsoft/CodeBERT.git>, 2023.
- [56] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. 02 2017.
- [57] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In IEEE, editor, *SEC&P 2021, 42nd IEEE Symposium on Security and Privacy, 23-27 May 2021 (Virtual Conference)*, 2021. © 2021 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
- [58] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1733–1750, Santa Clara, CA, August 2019. USENIX Association.
- [59] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*. ACM, May 2016.
- [60] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168, 2016.

- [61] SamMobile. Download firmware updates for your samsung mobile phone and tablet, Aug 2019.
- [62] Yuru Shao, Qi Alfred Chen, Z. Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *Network and Distributed System Security Symposium*, 2016.
- [63] Skylot. jadx: Dex to java decompiler, 2024.
- [64] steadfasterX. Steadfasterx/salt: Salt - [s]teadfasterx [a]ll-in-one [l]g [t]ool, 2024.
- [65] Xiaoyu Sun, Xiao Chen, Li Li, Haipeng Cai, John Grundy, Jordan Samhi, Tegawendé Bissyandé, and Jacques Klein. Demystifying hidden sensitive operations in android apps. *ACM Trans. Softw. Eng. Methodol.*, 32(2), mar 2023.
- [66] 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.
- [67] 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.
- [68] Bangrui Wan, Shuang Dong, Jianjun Zhou, and Ying Qian. Sjbed: A java code clone detection method based on bytecode using siamese neural network. *Applied Sciences*, 13(17), 2023.
- [69] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. One adapter for all programming languages? adapter tuning for code search and summarization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 5–16, 2023.
- [70] Min Wang, Pengcheng Wang, and Yun Xu. Csharp: An efficient three-phase code clone detector using modified pdgs. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 100–109, 2017.
- [71] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, page 1066–1077, 2018.

- [72] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1066–1077, 2018.
- [73] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.
- [74] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *International Joint Conference on Artificial Intelligence*, 2017.
- [75] 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, CCS '13*, page 623–634, New York, NY, USA, 2013. Association for Computing Machinery.
- [76] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. LibScan: Towards more precise Third-Party library identification for android applications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3385–3402, Anaheim, CA, August 2023. USENIX Association.
- [77] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. Detecting semantic code clones by building ast-based markov chains model. *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.
- [78] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. How android developers handle evolution-induced api compatibility issues: A large-scale study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 886–898, New York, NY, USA, 2020. Association for Computing Machinery.
- [79] 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.
- [80] Xian Zhan, Lingling Fan, Sen Chen, Feng Wu, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications, 2021.

- [81] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: the curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 1663–1674, New York, NY, USA, 2016. Association for Computing Machinery.
- [82] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019.
- [83] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. Libid: reliable identification of obfuscated third-party android libraries. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.
- [84] 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, CCS '18*, page 1165–1178, New York, NY, USA, 2018. Association for Computing Machinery.
- [85] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. Detecting third-party libraries in android applications with high precision and recall. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 141–152, 2018.
- [86] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3649–3666, 2021.
- [87] Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.
- [88] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.
- [89] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Con-*

*ference on Data and Application Security and Privacy, CODASPY '13*, page 185–196, New York, NY, USA, 2013. Association for Computing Machinery.

- [90] 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.
- [91] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012.
- [92] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. Ccgraph: a pdg-based code clone detector with approximate graph matching. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–942, 2020.

# APPENDICES

# Appendix A

## A.1 Our Solution

Table A.1: Paths identified in AOSP `getUidStats(..)`

Path ID	Instructions	Path Class
1	<code>getCallingUid()!=1000</code> <code>getCallingUid()!=uid</code> <code>return -1</code>	Failed AC
2	<code>getCallingUid()==1000</code> <code>return nativeGetUidStat(..)</code>	Core
3	<code>getCallingUid()==uid</code> <code>return nativeGetUidStat(..)</code>	Core

Table A.2: Paths identified in AOSP `getUidStatsWithoutCheckUid(..)`

Path ID	Instructions	Path Class
1	<code>return nativeGetUidStat(..)</code>	Core

Let us see how our solution detects Replicas and non-Replicas discussed in [Figure 3.1](#). The APIs `setPowerMode` and `setPowerModeChecked` are exact copies of each other and hence their program paths will also be identical and they will be identified as Replicas. For APIs `getUidStat` and `getUidStatsWithoutCheckUid`, we refer to [Table A.1](#) and [Table A.2](#) for their corresponding extracted program paths. We discuss how we extract these paths in detail in [section 4.1](#). We notice that Path # 1 in [Table A.1](#) corresponds to a failed access control – hence, it will be filtered out. The rest two paths are considered core logic due to their similarity to the API name. We denoise instructions pertaining to the UID-specific access control enforcement thus, the two paths are collapsed into a single core-logic path – which is an exact copy of the path extracted from `getUidStatsWithoutCheckUid`.

## A.2 Jaccard Similarity

```
1 public int getStreamVolumeForDevice(int i, int i2) {
2     return getStreamVolume(i, i2);
3 }
4
5 public int getStreamVolume(int i, int i2) {
6     int indexDividedBy10;
7     int index = this.mStreamStates[i].getIndex(i2);
8     if (this.mStreamStates[i].mIsMuted) {
9         index = 0;
10    }
11    if (index != 0 && mStVolAlias[i] == 3 && isFixVolDev(i2)) {
12        index = this.mStreamStates[i].getMaxIndex();
13    }
14    indexDividedBy10 = getIndexDividedBy10(index, i);
15    return indexDividedBy10;
16 }
```

Listing A.1: getStreamVolume and getStreamVolumeForDevice in AudioService

```
1 public String getContainerName(int i) {
2     String containerNamePerTypes;
3     if (isSecureFolderIds(i)) {
4         containerNamePerTypes = getSecureFolderName();
5     } else {
6         contNamePerTypes = getContNamePerTypes(getUsrManager(), i);
7     }
8     return contNamePerTypes;
9 }
```

Listing A.2: getContainerName in PersonaManagerService

```
1 public String getKnoxSettingsCustomName(int i) {
2     String containerName = getContainerName(i);
3     String string = getString(containerName);
4     if (string != null) {
5         return string;
6     }
7     return containerName + " settings";
8 }
```

Listing A.3: getKnoxSettingsCustomName in PersonaManagerService

**Example.** Consider the cases illustrated in [Listing A.2](#), [Listing A.3](#) and [Listing A.1](#). As shown, the API `getKnoxSettingsCustomName` invokes `getContainerName` to get the

name of container from Knox ID. It then proceeds to perform an additional functionality (retrieve the custom name). By considering the total cardinality of the sets, the Jaccard similarity is less than 0.8, although `getContainerName` execution path is a proper subset of `getKnoxSettingsCustomName`. In contrast, `getStreamVolumeForDevice` contains exactly one execution path - invoking `getStreamVolume` - therefore, it is a proper superset of execution paths from `getStreamVolume`. This clearly results in a Jaccard similarity, which is greater than 0.8.