Cues, Clones, and Cars: Access Control Issues in Customized Android

by

Parjanya Vyas

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Adwait Nadkarni Class of 1953 Associate Professor, Computer Science William & Mary
Supervisor(s):	N. Asokan Professor, Cheriton School of Computer Science University of Waterloo
	Yousra Aafer Associate Professor, Cheriton School of Computer Science University of Waterloo
Internal Member:	Urs Hengartner Associate Professor, Cheriton School of Computer Science University of Waterloo
	Meng Xu Assistant Professor, Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Mahesh Tripunitara Professor, Electrical and Computer Engineering University of Waterloo

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

Chapter 5 is adapted from a paper co-authored with Asim Waheed, Yousra Aafer, and N. Asokan [128], which was presented at the Usenix Security Symposium 2023. As the lead author of this paper, I designed, implemented, and evaluated the system (Bluebird). I also created the proof-of-concept exploits and managed their responsible disclosure. Asim Waheed evaluated different NLP techniques and implemented the best-performing model (BERT Cased). He also contributed to the NLP components used in Bluebird's evaluation. I led the writing of the paper with contributions from all authors.

Chapter 6 is adapted from a paper co-authored with Haseeb Ur Rehman Faheem, Yousra Aafer, and N. Asokan [127], presented at the Usenix Security Symposium 2025. As the lead author of this paper, I designed, implemented, and evaluated the system (Ariadne). Haseeb Ur Rehman Faheem created automation scripts for evaluation, created proof-of-concept exploits, and managed their responsible disclosure. I led the writing of the paper with contributions from all authors.

Chapter 7 is adapted from a paper co-authored with Syeda Mashal Abbas Zaidi, Shahpar Khan, and Yousra Aafer [143], presented at the International Conference on Automated Software Engineering 2024 (ASE '24). As a co-author of this work, I aided the design and implementation of the semantic replica detection and static analysis components, automated large-scale decompilation of ROMs, performed comparative evaluations, and analyzed the evolution of Replica access control enforcement. Syeda Mashal Abbas Zaidi and Shahpar Khan, as lead authors, led the design and implementation of the core system, performed parts of the evaluation, and led the writing with contributions from all authors.

Chapter 8 is adapted from a paper co-authored with Jumana and Yousra Aafer [75], presented at the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment 2025 (DIMVA '25). As a joint lead author of this paper, I identified and formalized the challenges, implemented the static analysis components, created some of the proof-of-concept exploits, and jointly led the evaluation, and writing. Jumana co-designed the system (AutoAcRaptor), implemented the similarity detection module, performed evaluation experiments, and prepared proof-of-concept exploits. All authors contributed to the writing of the paper. This material has been Reproduced with permission from Springer Nature.

Abstract

Android's open-source design and extensive customization have fueled its dominance across smartphones, automotive systems, wearables, and other domains. This flexibility, however, introduces serious security challenges, particularly in the enforcement of access control. Prior research has investigated inconsistencies within the framework, across layers, and across Android versions, yet important gaps remain, especially in detecting vendor-introduced data-driven customizations, replicated APIs, and platform-specific adaptations (e.g., automotive) that are difficult to capture with existing techniques.

This dissertation investigates how Android contextual features can be systematically leveraged to uncover access control vulnerabilities that evade prior analyses. It presents four main contributions:

- **Bluebird:** a probabilistic inference framework that derives access control requirements from application-side sensitivity indicators (UI cues and app-side access control). By fusing NLP-driven signals with static analysis, Bluebird identifies APIs whose protections do not match implied sensitivity. Applied to 14 ROMs, Bluebird flagged 391 likely under-protected private APIs.

- **Ariadne:** a static-analysis based technique built around a novel access control dependency graph abstraction that models explicit and inferred access control relationships among framework data holders. Ariadne detects inconsistencies introduced by data-driven vendor customizations that traditional tools miss. Evaluated on AOSP and vendor ROMs, it discovered 30 unique inconsistencies and enabled 13 proof-of-concept exploits.

- **RepFinder:** a large-scale measurement pipeline that identifies duplicated or "Replica" APIs created via copy-paste editing and evaluates their access control enforcement. Analyzing 342 ROMs from 10 vendors, RepFinder found replication to be widespread ($\approx$141 Replicas/ROM on average) and that a significant fraction ($\sim$37% on average) of Replicas are under-protected.

- **AutoAcRaptor:** a platform-specific static analysis framework for Android Automotive OS (AAOS) that identifies automotive entry points and evaluates both access control and feature-check enforcement. Applied to 10 AAOS ROMs, AutoAcRaptor reported an average of 23 auto feature and access control anomalies per ROM.

Collectively, these contributions show that Android contextual features such as app-side sensitivity indicators, framework data holders, and platform-specific service registrations can be systematically harnessed to reveal overlooked access control vulnerabilities. They also demonstrate that techniques for identifying framework customization-induced vulnerabilities can be adapted to emerging Android-based platforms such as Android Automotive OS by accounting for platform-specific differences.

Beyond these immediate contributions, this work opens two broader research directions. First, the contextual features explored in this work may not be exhaustive. Future research should aim to identify additional contextual signals—potentially through automated discovery—and explore an integration framework that makes it easy to incorporate new analyses into a unified solution. Second, the adaptation of these techniques to other Android-based platforms remains an open challenge. While AutoAcRaptor demonstrates feasibility for Android Automotive, other platforms such as Android TV, Wear OS, and Android XR present unique differences that require dedicated investigation to determine how well these methods generalize and what extensions are needed.

Acknowledgements

The list of people who supported me in many, many different ways, which eventually made this dissertation possible, is very long. So, I would like to start by first thanking everyone who made it possible for me to reach this point in my life. If I have missed someone, please know that it is not intentional, and I apologize sincerely in advance.

I am profoundly thankful to my parents, Dr Hitesh Vyas and Dr Gauravi Vyas, whose values and determination have been my greatest source of inspiration. You are, and always will be, the reason I am what I am, and I will never be able to thank you enough.

I am also thankful to my brother Malay, soon to (hopefully) be Dr Malay Vyas, whose thoughtful discussions and steady support helped keep me grounded and sane throughout this journey. You have always been someone who understood this journey, with all its joys and pitfalls, and without you I do not think I would've had the guts to keep going.

Next, I am very grateful to my son Mayas, whose laughter, joy, and boundless affection have been a continual source of happiness and strength. Your innocent smiles have always been the shining ray of hope that I so desperately needed in the darkest of times.

Most of all, I am thankful to my wonderful wife Kinjal, whose love, patience, and unwavering belief in me have been my greatest motivation. I am 100% sure, I would've left this journey a long, long time ago, if it was not you pushing me to go onward. You have been the one force of nature that has always, always kept me going in the right direction, motivated me never to despair, and pushed me to complete what I started. Thank you!

And how can I ever forget the goofiest member of my family? Thank you BarkUs, my golden retriever friend, that you exist in my life as a pure innocent soul full of joy, energy and never ending enthusiasm.

I would also like to thank my amazing friends at the University of Waterloo, with whom I have shared countless moments of learning, laughter, and collaboration. Your support, whether in tackling challenging research problems or simply being there during difficult times, has meant the world to me.

Last, but most certainly not the least, I am deeply, deeply grateful to both my supervisors, Dr N. Asokan and Dr Yousra Aafer, for their continuous guidance, insightful feedback, and steadfast encouragement, without which this work would never have seen the light of day. I hope that I have been able to learn at least a tiny bit of your amazing research and writing skills, and that you are (even if it is just a little bit) proud of what I have been able to achieve and deliver.

Dedication

To my family — for your endless love and faith in me.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AAOS Android Automotive OS

AOSP Android Open Source Project

CFG Control Flow Graph

DAC Discretionary Access Control

IPC Inter-Process Communication

MAC Mandatory Access Control

OEM Original Equipment Manufacturer

PDG Program Dependence Graph

ROM Read Only Memory (Firmware extracted from the read only memory of the device)

Chapter 1

Introduction

The Android operating system, with its open source nature and extensive customization capabilities, has fueled innovation and broad adoption across a diverse range of devices. This very flexibility, however, has introduced significant challenges to its security. A critical area of concern lies in the widespread customization of Android codebase by Original Equipment Manufacturers (OEMs), which frequently results in inconsistencies, often leading to vulnerabilities in access control mechanisms. These inconsistencies arise due to discrepancies in the access control enforced across different layers of the Android system, potentially exposing sensitive device functionalities to unauthorized applications.

These challenges manifest in several distinct ways. Access control vulnerabilities in Android can be broadly categorized into:

1. **Version and vendor inconsistencies:** discrepancies in how access control is enforced across different Android versions or OEM-customized ROMs.
2. **Intra-framework inconsistencies:** inconsistent access control across different code paths within the framework that access the same protected resource.
3. **Cross-layer inconsistencies:** inconsistent access control between layers of the Android stack (e.g., native vs. framework vs. application).
4. **Domain-specific inconsistencies:** access control issues due to Android's adaptation beyond smartphones (e.g., Android TV, Wear OS, Automotive).

Prior work has primarily focused on the first category, and to a limited extent on the second and third. This dissertation addresses unexplored gaps in categories 2 and 3, particularly

those introduced by OEM customizations. Category 4 remains largely unexplored: while Android TV has been studied [3], other domains such as Android Automotive and Wear OS have seen little systematic security analysis. This dissertation makes progress toward closing this gap by analyzing Android Automotive, while leaving the broader exploration of additional Android variants to future work.

This dissertation systematically investigates access control issues in customized Android frameworks by leveraging unique Android contextual features. The research centers around two key questions: (1) How can access control vulnerabilities caused by framework customization be systematically identified within the Android ecosystem? (2) How can these techniques be adapted and applied to emerging Android platforms, such as AAOS, which introduce new complexities due to their architectural differences such as the addition of car-specific services, and the lack of formal specifications for car-specific security and feature checks?

The work presented spans four major contributions, each grounded in novel static analysis techniques tailored to the unique features being leveraged.

First, Bluebird (Chapter 5) infers security specifications of framework APIs primarily used by preloaded apps. Rather than solely analyzing the framework's internal access control enforcement, Bluebird infers access control by modeling app-side behavior. Particularly, it models subtle sensitivity cues embedded in UI components and invocation patterns, called *sensitivity indicators*. To navigate the inherent uncertainty in linking app behavior to implied API sensitivity, Bluebird adopts a probabilistic inference framework that fuses NLP-driven insights and static code analysis. This approach reveals access control gaps where framework protections fall short of the sensitivity implied by app-side sensitivity indicators. Applied to seven Android Open Source Project (AOSP) and seven custom ROMs, Bluebird uncovered 391 private APIs with likely gaps in access control that included changing security policies for Samsung's SecureFolder, altering app-specific security settings in Samsung's PersonaManagerService, and manipulating audio related functionality in Amazon.

Second, Ariadne (Chapter 6) addresses another source of access control inconsistencies – arising from data-driven customizations – in the Android framework. Ariadne focuses on the addition or alteration of data holders (which are framework variables exposed via public APIs and protected by access control) by OEMs. These changes often miss enforcing access control, implied by the semantics of APIs or Java objects involving these data holders, making them difficult to detect by traditional analyses. To detect such issues, Ariadne introduces the Access Control Dependency Graph, a unified abstraction that models both explicit and inferred access control relationships among data holders using deterministic

and non-deterministic edges. By propagating access control requirements through this graph, Ariadne systematically uncovers inconsistencies, revealing a class of vulnerabilities that existing techniques like ACMiner [64], Poirot [35], and Bluebird [128] cannot detect. Applied to two AOSP and 11 vendor ROMs, Ariadne discovered 30 unique data-driven inconsistencies, and enabled the construction of 13 proof-of-concept exploits demonstrating its real-world impact.

Third, Chapter 7 presents RepFinder, a tool that identifies duplicated or replicated Android APIs — termed "Replicas" — introduced through copy-paste-editing of AOSP and OEM code by vendors. RepFinder detects syntactic and semantic similarity among core functional paths of APIs to highlight Replicas and systematically audits their access control enforcement. The study reveals that Replicas are prevalent across hundreds of ROMs and frequently introduce under-protected APIs, underscoring the security risks stemming from unnecessary code cloning and bloat. Empirical results from analyzing 342 ROMs across 10 vendors found 141 Replicas per ROM on average, with 37% (on average) identified as under-protected.

Fourth, extending the investigation beyond traditional mobile Android platforms, Chapter 8 addresses the security implications of Android's adaptation to automotive environments through AAOS. It presents AutoAcRaptor, a new approach, to statically identify automotive-specific entry points and analyze their access control and feature enforcement for security anomalies. AAOS presents unique challenges due to its architectural differences from traditional phone-based Android (especially in defining, registering, and using car-based framework services). AutoAcRaptor's consistency analysis uncovers auto feature and access control anomalies across automaker-specific AAOS ROMs, some of which translate into vulnerabilities, highlighting the pressing need for rigorous security evaluation of emerging Android-based domains. In a study of 10 AAOS ROMs, AutoAcRaptor identified 23 auto feature and access control anomalies on average per ROM, demonstrating both the tool's effectiveness and the prevalence of overlooked security risks in automotive Android customizations.

Together, these contributions provide systematic methods to identify inconsistencies caused by code duplication, framework-level customization, and domain-specific adaptation. Collectively, this work demonstrates that *Android contextual features offer unique insights that enable the systematic discovery of new types of access control vulnerabilities in traditional and emerging Android-based platforms.*

The dissertation provides affirmative answers to both research questions: (1) It presents systematic techniques to identify framework customization-induced access control vulnerabilities within the Android ecosystem, covering gaps left by prior work; and (2) it shows that

some of the inconsistency analysis techniques can be adapted to an emerging Android-based platform – AAOS. The broader exploration of other Android variants (e.g., Wear OS) remains an open challenge, left for future work.

In the following chapters, the dissertation presents the technical foundations, methodology, results, and implications of these contributions, demonstrating how Android's flexibility necessitates context-aware static analysis based frameworks to uncover and mitigate access control issues.

Chapter 2

Background

The challenges highlighted in the previous section cannot be fully understood without first examining how Android enforces access control, how prior work has attempted to analyze these mechanisms, and what abstractions this dissertation builds upon. This section provides that foundation. It begins with an overview of Android's architecture and its access control model, discusses the evolution of Android into AAOS and its implications for access control, then summarizes prior research on automated access control analysis, and finally describes the probabilistic and graph-based reasoning frameworks that underpin the scientific contributions of this dissertation.

2.1 Android Architecture and Access Control Model

Android is organized into three main software layers stacked on top of each other: the Linux kernel, the framework, and the application layer (Figure 2.1). Each layer provides abstractions to the one above it, while enforcing access control policies to protect sensitive resources. The access control enforcement is crucial as any underlying inconsistencies (in and across the layers) can lead to vulnerabilities.

2.1.1 Linux Kernel

The kernel, mostly written in C/C++, provides hardware abstractions and core functionality such as networking, display, audio, and camera support. Access control is enforced through two mechanisms: (i) traditional Discretionary Access Control (DAC) based on Linux

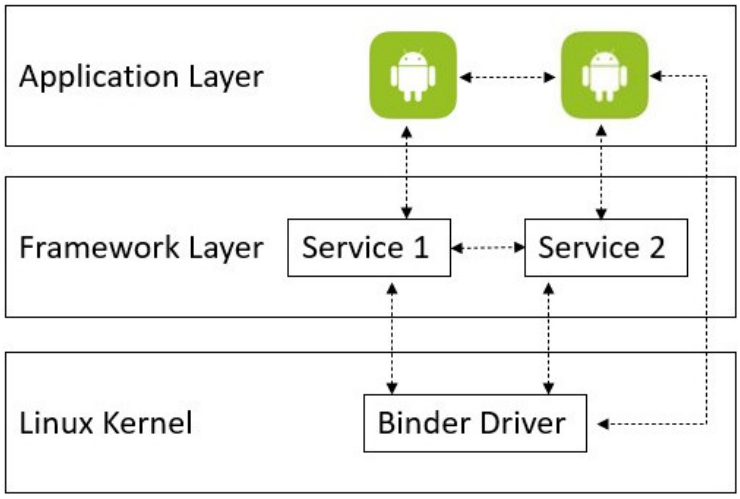

Figure 2.1: Android OS Architecture. The bi-directional arrows show interaction between various components through data/control flow.

UIDs and GIDs [51] assigned to apps; and (ii) SEAndroid [57], a Mandatory Access Control (MAC) system derived from SELinux, which provides fine-grained, least-privilege enforcement. Together, these mechanisms form the foundation of Android's security model.

2.1.2 Framework Layer

The framework, mostly written in Java, exposes functionality of the system services via APIs, used by applications through Android's binder-based Inter-Process Communication (IPC) [53]. This layer is the heart of Android's access control model. Its primary mechanism is *permissions* [56], which gate access to sensitive resources such as GPS, notifications, or system settings. Permissions are categorized into four protection levels: Normal, Dangerous, Signature, or System, reflecting different sensitivity and granting requirements [55]. Other checks such as user and process IDs, are used to enforce access control based on the physical user or calling app process identifiers.

2.1.3 Application Layer

At the top of the software stack are applications, mostly written in Kotlin and Java. Apps are categorized into system apps (which are part of the system image), pre-installed apps

```
// Create Car instance
Car mCar = Car.createCar(context);
// Create CarPropertyManager instance
CarPropertyManager pm = mCar.getCarManager(Car.PROPERTY_SERVICE);
// Invoke public APIs in CarPropertyManager
pm.getProperty(propertyId, zoneId);
pm.getPropertyList();
```

Figure 2.2: Invoking car APIs in AAOS.

(third-party apps that come pre-installed in the factory image), and third-party apps (installed by users through various external sources such as Google Play Store [60], or other app stores [110, 9]).

Apps enforce two main types of access control mechanisms. First, they declare permissions in their manifest [52] to restrict access to sensitive components. Second, they employ UI-based indicators such as warning dialogs, multi-step confirmations, or natural language cues, that signal the sensitivity of functionality exposed through the interface.

2.2 Evolution of Android to AAOS

In addition to smartphones and tablets, Android has also evolved to support other platforms such as Smart TV [3], Wear OS [62] and Automative [100]. Among these, AAOS has gained particular prominence. Unlike Android Auto, which merely projects phone content, AAOS extends the base Android platform to natively support automotive functions such as navigation, entertainment, and system control. This evolution has been realized primarily through two mechanisms: introducing new car-specific system services and customizing existing Android services.

Car-specific services. AAOS defines a dedicated *car-lib* library [25], which integrates new *Manager* classes (public APIs), a central `CarService` that registers and manages car-specific system services (e.g., `CarPropertyService`, `CarPowerManagerService`), and new car-specific permissions declared in *automotive_android_manifest.xml*. Apps access these services via Binder IPC, as illustrated in Figure 2.2.

```
public void setLocationEnabled(boolean locationEnabled) {
    if (mIsAutomotive && !locationEnabled) {
        // Calls to disable location are ignored in  automotive builds
        return;
    }
    mInjector.binderWithCleanCallingIdentity(() -> {
       getLocationManager().setLocationEnabledForUser(locationEnabled);
    }
}
```

Figure 2.3: Simplified implementation of `DevicePolicyManager.setLocationEnabled`.

```
public void lockNow(int flags,String callerPackageName,boolean parent){
    mUserManager.evictCredentialEncryptionKey(callingUserId);
    if (!mIsAutomotive) {
        // Power off the display ONLY on non-automative builds
        mInjector.powerManagerGoToSleep(SystemClock.uptimeMillis(),
            PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN, 0);
        }
    mInjector.getTrustManager().setDeviceLockedForUser(userToLock,true);
    ...
```

Figure 2.4: Simplified implementation of `DevicePolicyManagerService.lockNow`

Customized traditional services. AAOS also modifies existing Android services, typically gated by automotive feature checks (e.g., `PackageManager.FEATURE_AUTOMOTIVE`). Such changes include (i) *functionality blocking*, where APIs are disabled or partially altered for automotive builds (Figure 2.3 and Figure 2.4), (ii) *access control modification*, where additional constraints are enforced (e.g., user must be `USER_SYSTEM`), and (iii) *API additions*, where new automotive-only APIs are added to existing services (Figure 2.5). These adaptations ensure automotive-specific requirements are integrated, but they also increase the complexity and potential for inconsistencies.

2.3 Techniques for Access Control Analysis

The complexity of Android's layered design and widespread OEM customizations has motivated the development of various access control analysis techniques, which can be

```java
// API in LockSettingsService
public long addWeakEscrowToken(byte[] token, int userId) {
    if (!mContext.getPackageManager().hasSystemFeature(FEATURE_AUTOMOTIVE)) {
        throw new IllegalArgumentException("only for automotive devices.");
    }
    // actual functionality
    ...
```

Figure 2.5: Simplified implementation of `LockSettingsService.addWeakEscrowToken`

grouped into three complementary directions, each targeting different types of access control inconsistencies as outlined in Chapter 1.

2.3.1 API–Access Control Mapping

Framework APIs are the primary entry points for applications. Deriving precise mappings between these APIs and the access control they enforce is essential to identify potential gaps. Official documentation is often incomplete, outdated, or inconsistent across versions and vendor ROMs, and hence it is not easy to infer the access control specification, which can introduce version- and vendor-specific inconsistencies (type 1). Tools such as Stowaway [40], PScout [12], Axplorer [15], and ARCADE [2] automate large-scale derivation of API–permission mappings using static or dynamic analysis. While these approaches provide valuable insight into version- and vendor-specific behavior, they do not systematically account for intra-framework (type 2) or cross-layer (type 3) inconsistencies, nor do they consider context-dependent sensitivity cues at the application layer.

2.3.2 Inconsistency Analysis

A major source of Android access control vulnerabilities arises from inconsistent enforcement across the software stack. These inconsistencies may occur *within the framework* (intra-framework), *across layers* (cross-layer), or *across versions and vendor ROMs* (across ROMs). The former categories (intra-framework and cross-layer) focus on detecting inconsistencies *within* a single ROM, while the latter (across ROMs) involves comparing *across* multiple ROMs—either from different vendors with the same Android version, or from different Android versions of the same device. Prior work has explored multiple strategies to uncover such inconsistencies, which can be broadly grouped into three complementary directions: (1) convergence-based, (2) probabilistic, and (3) other analyses.

2.3.2.1 Convergence-Based Techniques

Convergence-based techniques aim to identify inconsistencies by comparing multiple enforcement points that should ideally converge to the same policy. These methods analyze the enforcement logic within and across layers to detect divergences in access control behavior.

For example, Axplorer [15] and Kratos [111] use static analysis, whereas Dynamo [28] and FReD [73] uses dynamic analysis, to examine cases where multiple code paths within the framework access the same sensitive resource but enforce different access control checks, corresponding to intra-framework inconsistencies (type 2). Tools such as IACEFinder [152] extend this idea to cross-layer discrepancies (type 3), identifying mismatches between kernel-level and framework-level enforcement. These studies show that convergence failures frequently lead to exploitable vulnerabilities, highlighting the need for systematic analyses of enforcement logic across multiple dimensions.

2.3.2.2 Probabilistic-inference Techniques

Traditional analyses typically assume a deterministic mapping between APIs and enforced access control. In practice, however, these mappings are often uncertain: private APIs are undocumented, specifications evolve over time, and resource sensitivity depends on application or user context. Probabilistic approaches explicitly model this uncertainty to identify inconsistencies that deterministic techniques may miss.

Poirot [35], for instance, introduces probabilistic reasoning to model distributions over possible security policies, allowing it to detect intra-framework inconsistencies (type 2) with low false positives. By reasoning over likelihoods rather than fixed rules, probabilistic techniques can uncover subtle enforcement gaps. A key limitation, however, is their dependence on the modeled rules and observations: prior works often define narrowly scoped rule sets manually, relying heavily on domain knowledge and leaving other relevant probabilistic relations unmodeled.

2.3.2.3 Other Techniques

Beyond convergence-based and probabilistic approaches, researchers have also examined additional mechanisms that influence access control, such as custom permissions [124, 84]. Other efforts have applied machine learning techniques [125] (see Chapter 3) to identify access control inconsistencies. While valuable, these analyses typically target specific vulnerability classes or rely on substantial manual intervention.

2.4 Generic Reasoning Techniques

In addition to domain-specific analyses of Android access control, this dissertation leverages two general classes of reasoning techniques to tackle specific challenges in modeling contextual features (Chapter 5 and Chapter 6): probabilistic inference and graph-based flow propagation. These approaches are not specific to Android, but provide systematic frameworks for reasoning about uncertainty and dependencies in complex systems.

2.4.1 Probabilistic Inference

Probabilistic inference [65] provides a principled way to represent and reason over uncertainties. In probabilistic models, each fact or rule can be associated with a likelihood or confidence value, and inference is performed over the resulting distribution of possible worlds [29]. For example, consider an API that accesses a sensitive sensor but is only partially documented: a probabilistic model can encode observations from app behavior, API usage, or developer annotations as weighted facts. Queries over this model allows estimating the probability that the API requires a specific permission or should be restricted in certain contexts.

This approach is particularly useful when rules are incomplete or noisy [41]. It allows analysts to combine multiple weak signals such as partial code patterns into a coherent assessment of security properties.

2.4.2 Graph-Based Reasoning and Flow Propagation

Graphs are a natural abstraction for modeling dependencies among system components, data flows, or control flows. Flow propagation techniques [26] leverage this structure to systematically reason about how properties or constraints propagate through the system.

For instance, consider a resource (such as a sensitive file) and the set of APIs that can access it. By representing the system as a directed graph where nodes correspond to resources or APIs, and edges represent access paths or dependencies, one can propagate security annotations (e.g., required permissions, trust levels, or sensitivity labels) along the edges. This allows identifying points where access control is missing, inconsistent, or weakened due to indirect flows.

Graph-based reasoning is particularly effective in identifying indirect inconsistencies, where a vulnerability arises not from a single code path but from the interaction of multiple

paths across a system [105]. By combining graph structure with annotations and propagation rules, analysts can systematically explore all possible flows, uncovering subtle security gaps.

Chapter 3

Related Works

Research relevant to this dissertation spans several areas: analysis of Android access control, application and UI analysis, probabilistic inference for security reasoning, OEM customization hazards, code clone detection, and emerging studies of Android variants including AAOS. This section reviews each of these areas and situates the dissertation's contributions within them.

3.1 Android Access Control Analysis

Framework access control in Android has been extensively studied. Early work focused on deriving permission-to-API maps through static or dynamic analysis. Tools such as Stowaway [40], PScout [12], Axplorer [15], Arcade [2], and Dynamo [28] produced large-scale mappings of framework APIs to required permissions/access control. These efforts established a foundation for empirical study of Android's access control model, highlighting gaps and inconsistencies in official documentation. However, because these tools only construct mappings, they do not systematically detect cases where enforcement is incomplete or inconsistent.

A complementary line of research introduced *inconsistency analysis*, where the goal is to identify cases in which multiple paths to the same resource enforce different protections. Kratos [111] pioneered this approach using call-graph analysis to locate divergent checks. AceDroid [1] extended the idea by normalizing authorization checks to reduce false positives. ACMiner [64] moved further toward automation, mining authorization checks semi-automatically from large codebases rather than requiring manually curated

lists. Cross-layer tools such as IAceFinder [152] and FReD [73] examined discrepancies spanning the Java, native, and kernel layers, surfacing cases where enforcement differed across layers in the Android stack.

More recent work recognized that strict deterministic analysis is brittle in the face of undocumented APIs or evolving specifications. Poirot [35] reconceptualized inconsistency detection by introducing probabilistic reasoning, showing that reasoning over uncertainty can reveal vulnerabilities overlooked by deterministic methods. This trajectory shows an evolution from deterministic mappings toward more context-aware, uncertainty-tolerant analyses—precisely the space in which this dissertation situates its contributions.

The existing work, especially Poirot, shows how context-aware and uncertainty-tolerant analyses can be beneficial over more traditional approaches. However, the question still remained whether such contextual features can be systematically leveraged to identify access control inconsistencies introduced by different forms of Android customization. This dissertation provides a comprehensive answer by proposing *Bluebird*, which leverages UI-based sensitivity indicators, *Ariadne*, which uses framework data-holders and their implied access control relations, and *RepFinder*, which detects syntactic and semantic replicas, and uses them to identify access control gaps. Through these approaches, we demonstrate that contextual features can indeed be systematically harnessed to uncover access control inconsistencies that prior works failed to capture.

3.2 Application and UI Analysis

Parallel efforts have analyzed Android applications themselves to identify vulnerabilities. Static analysis tools such as FlowDroid [11], Chex [87], and IccTA [82] model event-driven and inter-component behaviors to detect privacy leaks, privilege escalation, or component hijacking attacks. Dynamic tools like AppSealer [148] monitor runtime behavior to detect overprivilege or patch hijacking vulnerabilities, while other measurement studies have focused on risks introduced by pre-installed or update apps. For example, FIRMSCOPE [36] analyzed firmware apps and Blázquez et al. [18] examined FOTA (firmware-over-the-air) update mechanisms, both revealing systemic risks in the app ecosystem.

Another thread of work has focused specifically on *UI analysis*, under the observation that sensitive operations are often signaled to users via the interface. Tools such as SUPOR [71], UiPicker [90], and icon-based classifiers [138] extract sensitive input fields or widgets by analyzing text labels, icons, or widget hierarchies. These insights demonstrate that UI-level signals often capture developer and user expectations about sensitivity, even

when framework-level enforcement does not. This observation directly motivated the use of UI cues as application-side sensitivity indicators in Bluebird.

3.3 Probabilistic Inference in Security

Probabilistic inference has become increasingly influential in security research as a means to explicitly model uncertainty. In Android, Poirot [35] demonstrated that probabilistic reasoning can overcome the brittleness of deterministic inconsistency analysis by significantly reducing the false positives. Beyond Android, probabilistic methods have been applied to diverse problems: LambdaNet [134] uses probabilistic inference for type inference in neural program reasoning; PRIMO [91] improves static analysis precision by reasoning probabilistically about uncertain program states; and other works infer latent program properties such as unit consistency [77]. Probabilistic reasoning has also been used in reverse engineering [118], bug localization [31], and vulnerability detection [140]. These studies highlight the versatility of probabilistic approaches in reasoning over incomplete or noisy observations. This dissertation leverages the same principle in Android access control, where uncertainty arises naturally from undocumented APIs, vendor customizations, and context-dependent sensitivity.

3.4 OEM Customization Hazards

Another broad strand of work investigates the hazards of OEM-driven customization. Studies have shown that pre-installed apps often introduce overprivileged or insecure behaviors [135], while invalid or inconsistent input validation in OEM-private APIs can open exploitable flaws [147]. Residual APIs left in ROMs despite being unused create additional attack surfaces [34]. ADDICTED [154] examined security issues in custom device drivers, and SEPAL [142] studied customized SEAndroid policies using NLP and ML models to detect deviations. Possemato et al. [99] conducted a longitudinal study of customized devices, revealing widespread violations of Google's compatibility guidelines and exposing systemic risks. These works collectively underscore the difficulty of regulating customization at scale. Ariadne [127] and RepFinder [143] directly build on these works by addressing access control anomalies caused by data-driven and replicated customizations in the framework itself.

3.5 Code Clone Detection in Android

Code cloning has been studied extensively in software engineering and has specific security implications in the Android ecosystem. DNADroid [24] used dependency graphs to detect clones across Android apps, while other efforts [155, 153] focused on detecting piggybacked malware that embeds malicious code into legitimate apps. Clone detection has also been used to identify third-party libraries: LibScout [14], ATVHunter [144], LibRadar [149], and related tools map libraries to app binaries for vulnerability tracking. A range of detection techniques have been explored, including token-based approaches such as SourcererCC [106], CCAligner [131], and CCLearner [83], as well as structural and semantic techniques.

While these studies have focused primarily on application code or third-party libraries, RepFinder extends the scope of clone detection to the Android framework itself. By analyzing Replica APIs created through OEM copy-paste edits, RepFinder uncovers a previously overlooked but impactful source of vulnerabilities: inconsistently secured framework entry points duplicated during customization.

3.6 Security Analysis of Android Variants

Beyond smartphones and tablets, Android has been adapted to other domains such as wearables (WearOS) and smart TVs (Android TV). These variants extend the base Android platform with domain-specific features and constraints, which in turn introduce new attack surfaces. Several works have investigated their security properties, providing insights into variant-specific risks.

3.6.1 WearOS

WearOS integrates wearable devices with paired smartphones. Tileria et al. [122] developed *WearFlow*, a static taint analysis framework for analyzing information flows across companion apps in WearOS. By modeling cross-device communication channels, the authors uncovered privacy violations in popular app pairs. Other studies [141, 85] analyzed the WearOS platform more broadly, highlighting issues such as inadequate isolation, overprivileged access to sensors, and weaknesses in permission enforcement.

3.6.2 Android TV

Android TV represents another variant, where the platform is adapted for large-screen environments and controlled primarily through remotes or voice input. Aafer et al. [3] conducted one of the first systematic security studies of Android TV, identifying serious vulnerabilities via log-guided fuzzing. Tileria et al. [121] examined a large corpus of Android TV apps, comparing them against mobile counterparts and revealing widespread tracking and identifier misuse. Other works [86, 13] have investigated the security of Android and smart TV apps, revealing that these devices, like smartphones, are prone to security flaws, privacy leaks, and exploitable vulnerabilities across various attack vectors, while highlighting the lack of publicly available datasets and systematic analyses for TV-specific applications.

3.6.3 AAOS and Automotive Security

Prior work has largely focused on system-level or app-level risks rather than framework internals. For instance, studies of third-party AAOS apps revealed inconsistent privacy practices [96], while systems such as PriDrive [68] and Pricar [93] proposed privacy-preserving data collection mechanisms. Broader surveys [94, 95, 20] have mapped attack surfaces and potential vulnerabilities in connected vehicles, including AAOS deployments.

Beyond AAOS specifically, automotive security research has highlighted risks in diagnostic tools, in-vehicle infotainment (IVI) systems, wireless integrations (Bluetooth, Wi-Fi, cellular) [21, 88], and CAN-bus protocols [38, 132, 102]. These studies emphasize the breadth of the automotive attack surface. In contrast, AutoAcRaptor [75] provides the first systematic, static-analysis-based study of access control and feature-check inconsistencies within AAOS frameworks themselves, bridging the gap between prior app/system-level studies and framework-layer vulnerabilities.

Summary. Collectively, prior research has significantly advanced Android security analysis across APIs, applications, UI, and OEM customizations, while also beginning to probe AAOS and other Android variants. However, important gaps remain in capturing contextual features such as app-side sensitivity cues, data-driven customizations, Replica APIs, and AAOS-specific entry points—gaps directly addressed in this dissertation.

Chapter 4

Problem Statement

Android relies on access control mechanisms to protect sensitive resources. Prior work has shown that inconsistencies, undocumented policies, and overlooked contextual factors result in exploitable vulnerabilities. Existing analysis techniques often assume deterministic access control mappings and rarely extend to emerging Android-based platforms such as AAOS, where new architectural elements introduce additional complexity.

This dissertation addresses these limitations by leveraging **Android-specific contextual features** to systematically uncover access control vulnerabilities. The central thesis is:

> *Android contextual features offer unique insights that enable the systematic discovery of new types of access control vulnerabilities in traditional and emerging Android-based platforms.*

To investigate this thesis, the dissertation addresses the following research questions:

- **RQ1:** *How can unique Android contextual features be leveraged to systematically identify unknown access control vulnerabilities within the framework?*

  Addressed through:

  - **Bluebird:** app-side sensitivity detectors (Chapter 5)
  - **Ariadne:** data-driven relations and their access control implications (Chapter 6)
  - **Replicas:** syntactic and semantic similarity in APIs (Chapter 7)

- **RQ2:** *To what extent can these techniques be adapted to analyze other emerging Android platforms such as Android Automotive, given their architectural differences?*

  Addressed through the **AAOS study**, which adapts these methods to automotive feature checks and access control (Chapter 8).

Part I: Access control vulnerabilities induced by Framework customization

To address the first research question (**RQ1**), this part introduces three complementary techniques—**Bluebird**, **Ariadne**, and **RepFinder**—each designed to use distinct Android contextual features. While they target different manifestations of customization-induced vulnerabilities, they share a common methodological theme: systematically capturing contextual cues that traditional inconsistency analyses overlook.

- **Bluebird** (Chapter 5) leverages *app-side sensitivity indicators*, such as UI elements and app-side access control, to infer when framework APIs are more sensitive than what their declared protections suggest. It addresses gaps where framework-level access control enforcement lags behind developer and user expectations.
- **Ariadne** (Chapter 6) leverages *framework data holders (i.e., access-controlled variables within the Android framework) and the implicit access control specifications they carry*. These specifications arise from Java-object semantics (e.g., protection implied by class or field relationships) and operation semantics (e.g., protection implied by the APIs that access or modify the data). By reasoning over these implicit specifications, Ariadne identifies inconsistently protected data holders that lead to security vulnerabilities.
- **RepFinder** (Chapter 7) investigates vulnerabilities caused by *code replication*. OEMs frequently copy, paste, and edit framework APIs, creating Replicas that are syntactically or semantically similar to existing APIs but diverge in their access control enforcement. RepFinder combines similarity analysis with security auditing to identify under-protected Replicas.

The next three chapters present each of these techniques in detail.

Chapter 5

Auditing Private APIs via App-Side Indicators

The contents of this chapter are adapted from the paper [128] which introduces Bluebird, a probabilistic inference based approach for auditing private Android APIs. It builds directly on the motivation and framing established in the previous chapter, where we argued that Android-specific contextual features can be systematically leveraged to uncover access control vulnerabilities missed by traditional inconsistency analyses. Bluebird embodies this idea by exploiting *app-side sensitivity indicators* as a source of security specifications.

5.1 Introduction

Android device manufacturers often define *private APIs* in the framework, which are typically invoked by preloaded apps to access custom functionality. Ensuring *complete mediation* along the path from a preloaded app entry point to a private API's functionality is a security-critical process that requires careful coordination of security specifications across the app and framework layers. Both parties, preloaded app developers and framework developers, should enforce similar security measures. This is a challenging task, particularly due to the lack of security specifications for private APIs. Preloaded apps may fail to protect access to an invoked private API in face of external requests, thus introducing *hijack-enabling flows*. An unprivileged (malicious) app can leverage such flows as a stepping stone to trigger the private API, thus sidestepping potential framework-side access control. This classic scenario – reflecting one type of complete mediation lapse has been widely studied in the Android literature [87].

However, complete mediation lapse in the reverse direction has been under-explored. When a preloaded app protects a private API (i.e., by enforcing a security check along its invocation site), the API itself should likely provide a similar protection. Such a lapse can arise when the app developer uses her expert insight of the context in which an API is invoked to judge its sensitivity to devise an app-side security check accordingly, while the API developer lacked such insight.

The above intuition shows that the "security specifications"[1] implied by app-side sensitivity indicators can provide valuable insights for auditing the access control implementation of private APIs to discover potential flaws. However, turning this seemingly simple intuition into a security auditing technique faces two significant challenges.

First, the audit procedure should be able to extract unconventional sensitivity indicators from Android app entries. While traditional access control extraction techniques are effective for handling *non-UI based app components*, such as Receivers and Services, they cannot tackle UI-based app entries, such as Dialogs. The latter category often includes *implicit* sensitivity indicators ranging from security-relevant UI blocks (e.g., warning Dialogs), explicit user interaction (e.g., clicking on a confirmation button), to functionality occlusion.

Second, the audit procedure requires understanding the relevance of a specific API to sensitivity indicators in an app. As preloaded apps may invoke various code pieces, each contributing differently to an app entry[2], an invoked API *is not necessarily* relevant to the indicators.

Third, addressing this latter challenge further requires considering the invocation context of a private API in an app. As framework-side access control may target data flows or may be dependent on supplied arguments, app-side security checks could vary accordingly.

Addressing these challenges is a highly uncertain process, as we cannot precisely deduce that a given (UI-based) app entry is sensitive, nor can we derive the exact target of an app entry's access control. Rather, we can mimic (uncertain) human-like reasoning by relying on various hints. For example, a human security analyst can speculate that an app dialog for changing volume is sensitive because it includes the sentence *"Listening at a high volume may damage your hearing"*. She decides to associate that app entry with the API *increaseVolume* thanks to the presence of a Slider UI-widget entitled *"volume"*. She is reassured that her conclusion is correct since setting of the volume level directly flows from

[1] Henceforth, we use the term "security specification" in a broad sense to refer to the security policy that can be inferred from access control enforcement or other sensitivity indicators, rather than to a formal, written, security specification in the traditional sense.

[2] An app component or a UI block.

the slider to the API. She can thus infer that the API *increaseVolume* is sensitive because she is skillful at fusing app-side hints.

To solve these challenges while accounting for the inherent uncertainty, we propose Bluebird, an automated auditing technique for Android APIs. Inspired by the recent application of probabilistic inference in Android security [35], Bluebird models the audit task as a probabilistic inference problem. Bluebird pairs human-like understanding of app-side logic (e.g., using NLP techniques) with statically-derived app-side program semantics to infer the likely sensitivity level of an API, solely from the perspective of the apps calling it. Specifically, for each API, we introduce a random variable that indicates the probability of the API being sensitive. It is initially assigned a *prior* probability, reflecting the sensitivity level implied by its access control implementation in the framework. Next, Bluebird analyzes preloaded apps and extracts relevant security-specific *evidence* and *clues*. The *evidences* are app-side observations that assign a security indicator to app-side entry points, while the *clues* are *less certain* human-like reasoning rules that guide the inference procedure. They are conditional on various contextual and execution properties and regulated by an implication probability; meaning that if a conditional clue holds, we can associate app-derived sensitivity to invoked APIs with a level of confidence corresponding to the indicated implication probability.

The priors and probabilistic constraints are fed to a probabilistic inference engine to compute *posterior* probabilities of individual APIs being sensitive. Bluebird then detects a gap in the API's access control if the posterior marginal probability is more than the prior – meaning that framework implementation assumes less sensitivity than what is seen from the app's perspective. Note that Bluebird does not indicate the actual missing access control, but rather indicates the presence of a *gap*, since it is difficult to map the inferred sensitivity to a concrete access control check.

We applied Bluebird on 7 Android AOSP ROMs (including versions 8.1, 9.0, 10, 11, 12 and 13), and 7 Android custom ROMs, from Samsung, Amazon, Oppo, Vivo, Xiaomi, and ZTE. Our evaluation on the AOSP codebases demonstrates the validity of our approach. The majority of (low-sensitivity) AOSP APIs are found to contain no gaps, based on their app-derived sensitivity. Our evaluation on the custom ROMs reveal 391 private APIs with likely gaps in access control. We investigate 10% of the total reported cases manually, and identified 11 likely vulnerabilities. We built proof-of-concept exploits (PoCs) for a few selected cases (for which we could understand the recovered code). Some of the PoCs can be carried out with no permission or only low-privileged normal permissions and lead to various consequences, including changing security policies for Samsung's SecureFolder, altering app-specific security settings in Samsung's Persona, and manipulating Audio related functionality in Amazon.

We claim the following contributions:

- We propose Bluebird, an automated probabilistic security audit technique for unprotected Android APIs that is capable of leveraging and fusing uncertain app-side security logic. (Section 5.3)
- We develop a set of probabilistic inference rules that allow pairing human-like understanding of app-side security logic with statically derived program semantic and artifacts to conduct the security audit procedure. (Section 5.4)
- Via extensive empirical evaluation on 14 different ROMs (Section 5.5), we demonstrate the effectiveness of Bluebird by identifying 11 specific vulnerabilities (Table 5.5), all of which have been acknowledged by the vendors involved, and 6 have been granted bug bounties. (Section 5.5, Section 5.6)

5.2 Motivation

In this section, we use an example to illustrate how app-inferred security specifications can help identify APIs with an access control gap and the challenges that need to be addressed in the process.

Consider the highly-simplified usage scenarios of Samsung's private API `setSecureFolderPolicy` in the preloaded app *SecureFolder*, depicted in Listing 5.1, Listing 5.2 and Figure 5.1. The private API is invoked in two different components, in the Broadcast Receiver `PolicyUpdateReceiver` and in the Activity `FolderContainer`. In the first scenario (Listing 5.1 and Listing 5.2), the API (along with other internal app methods) is called upon triggering the `onReceive` method of `PolicyUpdateReceiver`. Note two distinct access control checks: (1) the broadcast receiver is protected with a custom signature permission (Listing 5.1), implying that the receiver cannot be triggered by third-party (non-Samsung) apps; and (2) the path leading to the API is control-dependent on another security check (Line 6 in Listing 5.2), that mandates that the receiver can only be invoked by (physical) users with id >= 150 (on Samsung devices, these correspond to mobile device management admins).

In the second scenario (Figure 5.1), the API is triggered upon launching the Activity `FolderContainer`. Besides the userId check at Line 5, there is a more-difficult to detect access control: The user needs to enter her password in the Fragment (Figure 5.1.A,) , and click on *Continue* button before landing on the `FolderContainer` Activity. Observe that

although `FolderContainer` is exported (Figure 5.1.C), the access control check (Line 5 in Figure 5.1.D) implies that the user needs to be logged in as user id >= 150.

```
<receiver android:name=".common.policyagent.PolicyUpdateReceiver" android:
    permission="samsung.permission.RECEIVE_SECURE_FOLDER_POLICY_UPDATE">
```

Listing 5.1: PolicyUpdateReceiver Manifest Definition

```
public class PolicyUpdateReceiver extends BroadcastReceiver {
   int i = 150;
   if(!TestHelper.isRoboUnitTest())
     i = UserHandle.getCallingUserId();
   Log.d("[FOLDER].PolicyBootReceiver", "START_POLICY_UPDATE");
   if (i >= 150)
     PolicyHttpClient.downloadPolicy();
```

```
public class PolicyHttpClient {
   public static void downloadPolicy() {
     if (PolicyHttpClient.isRebootCase || PolicyHttpClient.isAppUpdateCase){
       ...
       xmlPullParser.setInput(assetManager.open(Policy_XML), null);
       while (true) {
         xmlPullParser.next();
         if ("DisallowPackage".equals(xmlPullParser.getName()))
           mDisallowList.add(xmlPullParser.getAttributeValue(null, "name"));
       }
       setSecureFolderPolicy("DisallowPackage", mDisallowList, userId);
       unInstallDisallowedApps();
       enableBluetooth(); ...
```

Listing 5.2: PolicyUpdateReceiver Code Listing

Given the observed consensus among the two entries, we can intuitively deduce that the API is *likely sensitive*, and hence its framework-side implementation should enforce an access control. Unfortunately, it does not. Consider the API's (simplified) implementation depicted in Listing 1 (extracted from Samsung SM-A3058):

```
public void setSecureFolderPolicy(String key, List pkgList, int user) {
   mSecureFolderPolicies.put(key, pkgList);
   saveSettingsLocked(user);
   if(key.equals("DisallowPkg"))
     setApplicationBlackList("DisallowPkg", user);
```

Listing 5.3: setSecureFolderPolicy

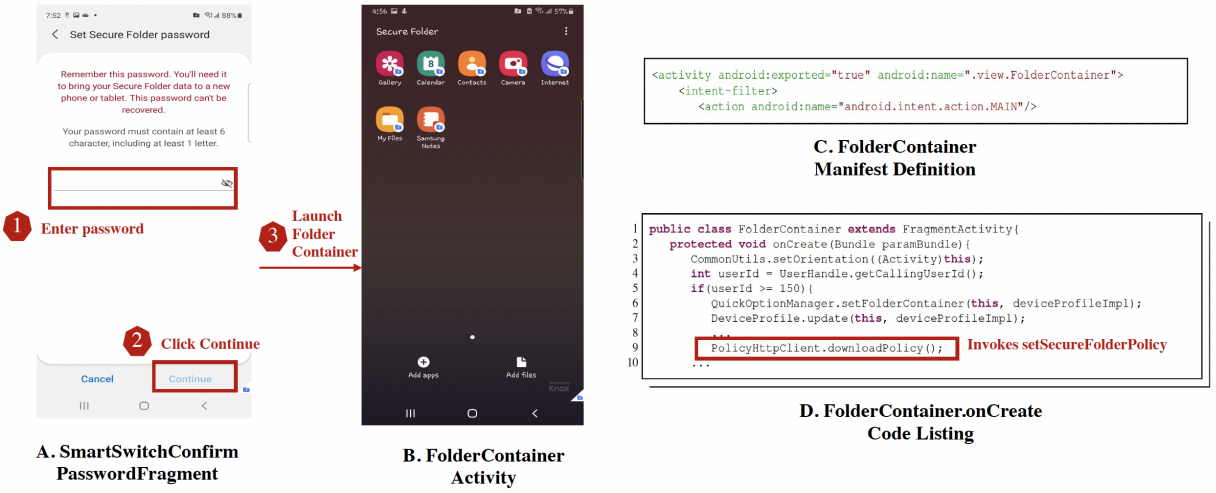

Figure 5.1: Folder Container Activity

The API allows overwriting internal Secure Folder policies for a given user and manipulating the Secure Folder's blacklisted apps. Given the sensitivity of these operations, we can confirm that our earlier deduction – based on the security specifications from the preloaded app, is *most likely* true. We built a PoC to exploit this vulnerability and demonstrated that it is possible for a third-party app to alter security policies of Samsung's Secure Folder container. Samsung has acknowledged and fixed the vulnerability.

Our goal is to develop a systematic procedure to leverage similar insights to audit framework APIs. It entails the following three challenges:

Challenge 1: Diverse App-Side Sensitivity Indicators. App-side sensitivity indicators feature substantial diversity: Apps use both (1) traditional access control enforcement (e.g., permissions, uid checks, and user checks), largely similar to that used in the framework, and (2) app-specific protection mechanisms. Some of them are explicit, imposed at the declaration point of app components, such as blocking access to the components (i.e., via setting the `android:exported` flag to false). Others are implicit, implemented by different UI-based *sensitivity indicators* (e.g, UI properties and interactions) like those in Figure 5.1.A. Observe that besides the password requirement in the Figure, the UI itself includes other elements indicating the sensitivity of the operation: (a) the text includes sensitive keywords, e.g., *recovered*, *password*, and (b) the password field is masked.

This diversity of app-side security indicators, particularly UI-based ones, renders their automatic extraction challenging.

Challenge 2: Estimating the Contribution of an API to an App Component. Components of preloaded apps tend to be quite complex, involving the invocation of code from different sources. Along the call site of a private API, a component may invoke other APIs, internal methods and external library calls in order to execute its overarching functionality. The auditing technique needs to differentiate among these pieces of code, separating *main contributors* from *auxiliary* ones.

We observe through our analysis that the degree of contribution is important for understanding the relevance of an invoked private API to the component's sensitivity indicators. Main contributors are *more likely* to be relevant than auxiliary contributors. Failing to account for this factor may lead to inaccuracies. For example, the audit process can conclude that an (insensitive) auxiliary API has an access control gap – thus overwhelming a security analyst with false alarms.

We note though that demarcating main and auxiliary code is difficult. While a human analyst can make the distinction using known hints (e.g., mnemonic beacons), a machine cannot as it requires understanding and analyzing intrinsically noisy code.

Consider the code listing of `PolicyUpdateReceiver` (Listing 5.2). We can deduce that the following are main contributors:

1. The internal method `downloadPolicy()` is a likely main contributor to the Receiver's functionality. It shares the common word *policy* with the Receiver's name.

2. `setSecureFolderPolicy` (line 11) is a likely main contributor to the private method `downloadPolicy()` as they share a common word *policy*. Give our observation in (1), we can transitively propagate our conclusion up to the Receiver: `setSecureFolderPolicy` is a likely main contributor to the Receiver's functionality.

3. We can reinforce our transitive conclusion in (2) by another observation. `setSecureFolderPolicy` and the Receiver's required permission (`..RECEIVE_SECURE_FOLDER_POLICY_UPDATE"`) (Listing 5.1) share a stronger naming similarity. Thus, we are more confident now that `setSecureFolderPolicy` is a likely main contributor to the Receiver's functionality.

4. The sdk API `enableBluetooth` and the private method `uninstallDisallowedApps` are likely main contributors to the calling method `downloadPolicy`. Although we cannot spot a syntactic naming similarity, we know that the words *enable* and *disallow* are related to the concept of *policy*. We can use similar transitive reasoning as in (2) to deduce that `enableBluetooth` and `uninstallDisallowedApps` are likely main contributors to the Receiver.

Conversely, the methods `isRobotUnitTest` and `getCallingUserId` are likely auxiliary contributors as they bear no similarity to the Receiver's name.

In summary, our reasoning concludes that the API `setSecureFolderPolicy` and the two sdk APIs, are *likely* more relevant to the Receiver's permission (i.e., `RECEIVE_SECURE_FOLDER_POLICY_UPDATE`) than the auxiliary contributors.

Reasoning scope: We limit our reasoning scope to code elements that are (directly or transitively) related to target APIs via call dependencies; that is, we only estimate the contribution scope of target private APIs and their (transitive) callers. This focused reasoning is driven by our domain-based hint that analyzing code unrelated to the APIs cannot enhance our belief in the API's contribution extent. For example, although we can conclude that the internal method `setFolderContainer` (at line 6, Figure 5.1.D) is a main contributor to `FolderContainer` Activity, it does not impact our conclusion regarding the contribution of the target private APIs (because a component may have many main contributors).

Challenge 3: Understanding the Invocation Context of an API in an App Component. We further observe through our analysis that, when serving external requests, an app entry may allow a varying execution extent of APIs. For instance, it may enable a full execution; i.e., it allows other entities[3] unlimited control of the API's parameters and a direct channel to read its execution output. Alternatively, it may allow a limited execution. That is, it allows other entities to trigger the API, but only under specific parameters, strictly controlled by the app itself (e.g., constant parameters, parameters read from files, or special APIs' return values). The execution output may be similarly constrained; the app entry does not disclose the results back to the external entity.

We note that an API's execution extent is important for understanding its relevance to its app-side sensitivity indicators. An app entry allowing a full execution, use sensitivity indicators reflecting that of the API's access control, while those allowing limited execution may not necessarily do so (i.e., an app-side sensitivity indicator may be lower than what is indicated by the API's access control.

We identified two main reasons leading to these cases:

- Data sensitivity: These cases include app entries invoking *data-sensitive* APIs; i.e., those setting or returning data (*sinks* and *source* APIs) The entries are *likely* to impose protection dependent on data flows. For example, if an app entry does not disclose a *source* API's returned value, it is unlikely to enforce a protection.

[3]Apps or the user.

- Path sensitivity: These cases reflect app entries that invoke APIs whose access control is dependent on provided arguments. The entries are likely to use different protections depending on the API's parameters).

This category of versatile APIs can impede our proposed auditing. Without resolving the context, a naive auditing technique may associate a *seemingly* relaxed access control with a potentially sensitive API. Consider the invocation site of `setSecureFolderPolicy` in Listing 5.2, line 11. The third argument *userId* constraints the policy changes to the scope of the calling user (i.e., `getCallingCurrentUserId`[4]); that is, a user can only change her own policy. The argument hence can be interpreted as an *implicit* security property, which should *likely* be enforced by the API implementation (e.g., by verifying if the argument matches a specific value).

Key Observations. From the above examples, we note that auditing framework APIs based on app-side security specifications requires reasoning about uncertainty. Besides, we note that app-side security specifications may include inherent *contradictions*, due to: (1) some preloaded app developers may over-protect APIs, and (2) others may not be able to judge the sensitivity of the undocumented private APIs.

Our Solution. Inspired by the recent application of probabilistic inference in Android security[35], we resort to modeling the audit task as a *probabilistic inference* problem. Probabilistic inference allows drawing conclusions from uncertain information and observations by modeling them in the form of a probability distribution. Specifically, we need to compute the marginal probability that an API is sensitive from various app-side hints. The probability depends on the sensitivity of the invoking app entries, and is conditional on various properties connecting the API to the entries.

In probabilistic inference, a random variable (a boolean variable with a probability) is used to denote a predicate (e.g., "an API is sensitive" or "an app entry is sensitive"). Inter-dependent predicates (e.g., "an API is likely sensitive if it is a main contributor to a sensitive app entry") are represented as probabilistic implication constraints, as follows: $pred_1 \xrightarrow{pr} pred_2$ where pr denotes a (prior or computed) confidence in $pred_1$ implying $pred_2$ to be true. The predicates may be involved in multiple constraints denoting their dependencies. Probabilistic inference aggregates the constraints, which are considered together to form a joint distribution, and computes a posterior marginal probability of the APIs' sensitivity. Intrinsically, the final probability reflects a better sensitivity estimation, as the impact of uncertainties and contradictions will be overpowered by dominant hints.

[4]userId is set to `getCallingCurrentUserId`, not shown for brevity.

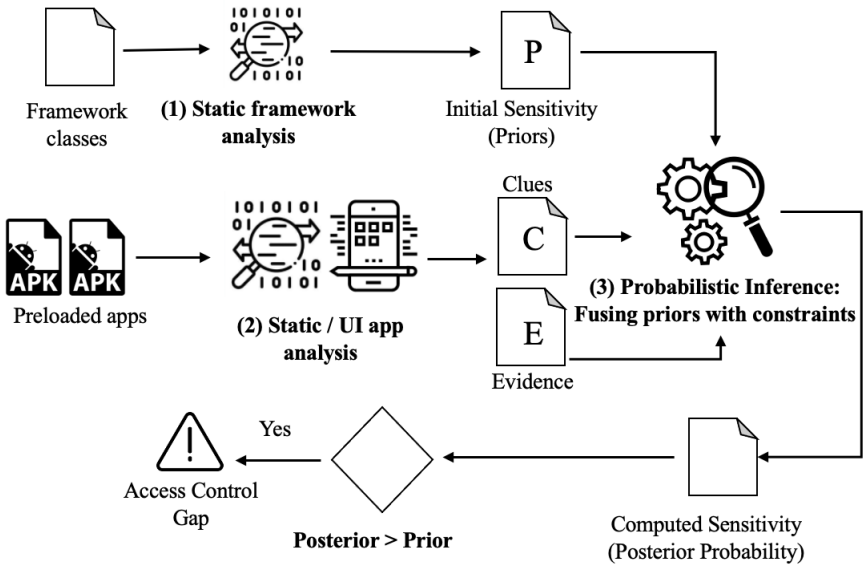

Figure 5.2: Bluebird architecture

Next, we discuss our technique in more details, and explain how it can be used to detect access control gaps.

5.3 Our Technique

Figure 5.2 presents a high level overview of our probablistic security audit system, Bluebird. It consists of three phases:

Phase 1 - framework-side analysis to determine priors: Bluebird begins by statically analyzing the framework-side to identify Android APIs and pertaining access control logic. For each framework API, Bluebird introduces a random variable that denotes the probability of the API being sensitive. The random variable is initially designated a sensitivity level, called *prior-probability*, which has two possible constant values: p_h denotes that the API is likely sensitive (because static analysis reveals that the API enforces access control), and p_l = 1-p_h denotes that it is not.

Existing literature [139, 77, 31] of probability inference leverage predefined prior probability values derived from domain knowledge and show that inference results are typically insensitive to these initial values. Here, we follow the same practice, by setting 0.9 for p_h and 0.1 for p_l.

Given our goal, Bluebird limits the audit procedure to APIs with an initial p_l prior-probability.

Phase 2: app-side analysis to extract evidences and clues: To facilitate auditing, Bluebird analyzes preloaded apps' code and extracts relevant *evidences* and *clues* as follows:

First, it collects *evidences* (i.e., facts collected through app analysis) which reflect sensitivity indicators adopted by app entries; e.g., traditional access control properties and UI signals. The collection requires extra processing of UI-based entries (e.g, via Natural Language Processing) to estimate its likelihood of being sensitive. Bluebird introduces a random variable for each app entry to denote its sensitivity. Depending on the nature of the evidence, entries may be assigned preset or computed prior-probabilities. For example, traditional access control-based evidence (e.g., permissions) are assigned preset values (e.g., p_h, p_l), while those requiring extra processing (e.g., NLP-based evidence) are assigned computed priors.

Second, Bluebird moves to collecting *clues* (i.e., implications that help connect evidences with invoked APIs); e.g., naming similarity between the app entrypoint and the API. Unlike *evidence*, *clues* are *less certain* human-reasoning rules that guide the inference procedure to derive a sensitivity level for an API based on its invocation in apps. Specifically, Bluebird looks for two kinds of *clues*, contribution extent and contextual features, which are designed to solve the challenges discussed in Section 5.2. *Clues* are conditional and are regulated by an implication probability; that is, if a property holds, Bluebird propagates an app-side sensitivity to invoked APIs with some probability. Note that the probability reflects the uncertain nature of the clues. For example, we can only state that *An API is likely sensitivity because it is highly relevant to the sensitivity indicator in an app entry with a (computed) confidence.*

Phase 3: fuse priors with probabilistic constraints to compute posterior probabilities: Bluebird leverages a probabilistic inference algorithm [78] to fuse the priors and probabilistic constraints to compute marginal posterior probabilities of individual APIs being sensitive.

Finally, Bluebird detects APIs with access control gaps if the posterior probabilities are more than the priors. The new probabilities reflect a better assessment thanks to the evidences and clues collected in the apps.

Motivating Example Walk-through. Next, we will walk the reader through the motivating example to illustrate how Bluebird audits Samsung's API `setSecureFolderPolicy`. Before probabilistic inference, Bluebird associates the API with a p_l prior probability – recall the API's framework-side implementation does not enforce access control.

Next, Bluebird analyzes the preloaded app *SecureFolder* (i.e., the invoking app), to derive a sensitivity-level for each app entry leading to the API. Based on traditional access control analysis, the Receiver `PolicyUpdateReceiver` (Listing 5.1 and Listing 5.2) is assigned a high sensitivity score p_h since it uses a system-level permission.

Based on NLP, UI and component chaining analysis (details are explained later in Section 5.4), the Activity `FolderContainer` is assigned a *computed* sensitivity score 0.99. Note that the high score is deduced from two evidences: (1) the physical user id (Line 5 in Figure 5.1.D) and (2) the Activity is reachable from a highly sensitive fragment (Figure 5.1.A).

Bluebird then leverages program analysis and NLP analysis to extract the probabilistic constraints as follows:

Contribution Extent. Bluebird constructs *contribution* probabilistic constraints to encode the speculation that an API is likely relevant to the sensitivity indicators in the app with a probability relative to its contribution extent. Accordingly, Bluebird proceeds by mimicking the reasoning of a human analyst as in Section 5.2 to estimate a contribution score for `setSecureFolderPolicy` to the Receiver – which yields a score = 0.75 (details explained later in Section 5.4). It then propagates the Receiver's sensitivity (p_h) to `setSecureFolderPolicy` with the implication probability: [p_h * Contribution score] = 0.71. We use the probability factor p_h to account for the uncertainty that developers may not always obey the naming conventions[5].

The constructed constraint suggests that the API is likely sensitive with a posterior probability 0.78 after inference. Constructing a similar constraint in `FolderContainer` Activity further increase the poster probability to 0.91 (Details are omitted for brevity). At this stage, we can conclude that the API is likely more sensitive than its prior.

Context Dependency. Bluebird constructs *context* constraints to account for contextual properties that may influence the propagation of the priors. For the motivating example, Bluebird recovers the following property: `setSecureFolderPolicy` takes a special type of input (i.e., `getCallingUserId`) at the two app entry points. As such, it formulates two probabilistic constraints, each propagating the entry's sensitivity to the API with high confidence (for a more elaborate explanation, refer to Section 5.4). Last, the inference procedure combines the priors and context constraints to project a final posterior marginal probability. The inference yields a final posterior = 0.94 implying a gap in the API's access control.

[5]Note that the factor does not penalize the calculation significantly.

Automated inference. Although we describe the inference procedure as individual steps, our tool encodes extracted evidences and clues as prior probabilities and conditional probabilities that are resolved automatically. The whole inference procedure is transparent and seamless to security analysts.

5.4 System Design

In this section, we dive into the details of Bluebird. We introduce the inference rules used by Bluebird in Table 5.1. The Table lists the predicates relevant to the rules, their symbols, definitions, and pertaining constraints.

5.4.1 Framework Analysis: Computing Priors

Bluebird starts by extracting framework-side security checks to estimate priors (i.e., initial sensitivity of APIs). For each API defined in the Android system services, it builds an interprocedural control flow graph and traverses it in a depth-first fashion to identify access control checks. It extracts permission, user, and process id checks by looking for corresponding enforcement methods, resolving relevant conditional statements, and tracking data flows to resolve arguments, if any. The analysis then normalizes the checks according to the scheme proposed by AceDroid [1]. Bluebird establishes the prior probability using Rule 1: an API *api* has p_l prior-probability of being sensitive if its framework implementation reveals a normalized access control corresponding at most to `Normal` level or to $\emptyset$[6].

5.4.2 App Analysis: Extracting Sensitivity-Relevant Evidences

Bluebird begins app analysis by identifying *direct* app-side entries leading to (private) APIs through reachability analysis. Here, we rely on an existing tool, EdgeMiner [19] to connect implicit control flows. The reachability analysis yields a mapping of app entries and the APIs they (eventually) invoke. We note that ensuring a comprehensive mapping is challenging, particularly for heavily obfuscated apps. A discussion of how obfuscation affects our analysis can be found in Section 5.7.

A cornerstone of this analysis is to model local Inter-Component Communication (ICC) in order to identify *indirect* app entries leading to an API. As such, Bluebird can propagate

[6] AceDroid uses standard permission protection levels for normalization.

Table 5.1: Predicate/random variable and constraint definition

Predicate	**Symbol**	**Definition**	**Constraints**
Sensitive	$S(ep\|api)$	Entry point ep or API api is sensitive	
[1] Weak Framework-side Access Control	$F(api)$	Framework API api enforces weak or no access control	$S(api) = 1(p_l)$
[2] Weak App-side Access Control	$F(ep)$	App entry ep enforces weak or no access control	$S(ep) = 1(p_l)$
[3] Strong App-side Access Control	$\neg F(ep)$	App entry ep enforces Strong access control	$S(ep) = 1(p_h)$
[4] UI Dialog type	$U(ep)$	App entry ep corresponds to a sensitive UI type	$U(ep) = 1(p_h)$ $U(ep) \xrightarrow{p_m\dagger} S(ep)$
[5] Non-Disposable Dialog	$D(ep)$	App entry ep is a non-disposable UI block	$D(ep) = 1(p_h)$ $D(ep) \xrightarrow{p_m} S(ep)$
[6] Multi confirmation UI	$C(ep)$	Entry ep enforces a double confirmation (UI)	$C(ep) = 1(p_h)$ $C(ep) \xrightarrow{p_m} S(ep)$
[7] Sensitive UI Text	$N(ep)$	App entry ep likely contains sensitive keywords	$N(ep) = 1(score)$ $N(ep) \xrightarrow{p_h} S(ep)$
[8] NLP-based Contribution	$T(ep, api)$	API api is a main contributor to ep based on NLP.	$S(ep) \xrightarrow{CT\ddagger * p_h} S(api)$
[9] Modifiable context i.e. full execution	$M(ep, api)$	api's context is fully modifiable through entry ep	$\neg S(ep) \xrightarrow{p_h} \neg S(api)$ $S(ep) \xrightarrow{p_h} S(api)$
[10] Non-modifiable context	$\neg M(ep, api)$	The context of API api is non-modifiable through entry ep	$\neg S(ep) \xrightarrow{p_l} \neg S(api)$ $S(ep) \xrightarrow{p_h} S(api)$
[11] User-modifiable context	$UM(ep, api)$	api's context is modifiable only through user input to ep	$\neg S(ep) \xrightarrow{p_m} \neg S(api)$ $S(ep) \xrightarrow{p_m} S(api)$
[12] Special Input	$P(ep, api)$	API api takes an input hinting implicit AC in ep	$\neg S(ep) \xrightarrow{p_l} S(api)$ $S(ep) \xrightarrow{p_h} S(api)$

†p_m is explained in Subsection 5.4.2.
‡ CT refers to the computed *Contribution* as explained in Subsection 5.4.3.

evidences from indirect callers to transitively invoked APIs. Consider the scenario where a password Activity A starts a non-exported Service B, which in turns invokes a private API. Our analysis should associate both A's and B's evidences with *api*. Bluebird models ICC by resolving explicit intent targets as follows: It identifies the arguments at Intent construction methods and extracts the target component name by backward tracking through their Def-Use chains. Intent resolution is thus limited to explicit constructions. Bluebird leverages the resolved ICC flows to build a *local* component call graph, which is then used to augment the initial mappings by adding the resolved indirect app entries.

Bluebird analyzes the recovered entries to collect two kinds of security evidences.

(a) Traditional Access Control Evidences. These represent (1) permission enforcement and exposure flags at the level of components definitions, and (2) programmatic access control implemented in the life-cycle methods of components. We use the same normalization scheme [1] to map a traditional access control evidence to a sensitivity level.

Bluebird leverages Rules 2 and 3 to formulate app-side evidence constraints. Rule 2 states that an entry *ep* is unlikely sensitive if it implements a weak (e.g., `Normal` permission) or no access control; while Rule 3 states that *ep* is sensitive if it implements a stronger one (e.g., `System`, `Dangerous` permissions, `User` security check, or uses a `non-exported` flag).

(b) UI-Block Sensitivity Indicators. These evidences represent UI app entries containing sensitivity indicators. For each UI-block, we automatically examine its XML layout and extract the following: type, identifiers, hints, and textual labels of defined elements. Since some elements can be dynamically constructed, we statically analyze the apps' bytecode to locate invocation to UI-blocks construction methods and resolve their arguments using def-use chains (e.g., we resolve the argument of `setText` method for a `TextField` UI-block). We then leverage the extracted information to estimate the sensitivity of a UI block according to the following rules:

- If a UI-block type corresponds to *known* sensitive types (e.g., `AlertDialog`, `android:inputType=textPassword`), we associate the UI-block with a high sensitivity level with a medium confidence p_m[7] – Rule 4.
- If a field attribute corresponds to a non-disposable type (i.e., cannot be dismissed), we similarly associate the UI-block with a high sensitivity with medium confidence – Rule 5.

[7] p_m is set to 0.8; The value was empirically determined.

- If the path leading to the invocation of the API from the UI-block contains more than one known UI based callbacks (e.g., multiple `onClick`), we associate the UI-block with a high sensitivity level with medium confidence – Rule 6.

Pretrained NLP Model for Sensitivity Prediction. Textual elements in a UI block can provide insights into the impact of an invoked API, its access control, and thus sensitivity level. However, we note that curating a list of sensitive keywords for the Android domain from the list of known keywords (e.g., *confirm*, *careful*) is not sufficient. For example, words like *reboot*, *device* and *boost* do not imply security-critical functionality in general usage, but do in fact hint an access to Android-specific privileged functionality.

To address this challenge, we build an NLP model that automatically maps free-form written Android UI descriptions to sensitivity levels. The model can hence be queried to check the sensitivity of a given new text under the Android context. Specifically, we leverage the observation that a large corpus of free-form text can be statically extracted from preloaded apps' UI blocks. Through static analysis, we can map a UI block and its relevant free-form text to invoked AOSP APIs. We can then automatically label the text with the proper sensitivity level by examining the invoked API's access control; i.e., if the API is sensitive, we automatically label the related textual elements as sensitive and vice versa.

Our analysis yielded 4516 labeled texts, with 4208 sensitive samples and 309 non-sensitive samples extracted from 698 preloaded apps. To address the class imbalance, we down-sampled the majority class (sensitive) by a factor of 2 yielding 2258 samples, and up-sampled the minority class (non-sensitive) by a factor of 1.5 yielding 464 samples, with the goal of improving model performance. The final model had an F1-score of 0.969.

Since our training corpus is relatively small, we use transfer learning to build the model. We start with a pre-trained BERT Cased model[37] and train it on our labeled data. This choice channels the knowledge learned by the pre-trained model to our target Android-specific domain. We used 1024 epochs with all layers unfrozen, and non-overlapping subsets of 20% data for validation and testing. Given the probabilistic nature of our analysis, we apply the softmax operation as the final layer in our NLP model to produce a sensitivity probability (the probability of an input to correspond to the sensitive class). Bluebird uses the produced probability (denoted as *score*) to formulate the constraint in Rule 7, Table 5.1. The constraint sets the prior of a UI-based app entry to *score* with high confidence.

5.4.3 App Analysis: Extracting Contribution Clues

As mentioned, we observe that an API is relevant to the sensitivity indicators in an app entry based on its contribution extent. Thus, we model this observation as a conditional inference constraint, or *clue*, that propagates an app entry's sensitivity to invoked APIs with confidence.

Rule 8 presents the program artifacts that we rely on to generate the contribution constraints. It reflects the knowledge that humans tend to rely on naming hints to identify the focal instructions in a program and that main contributors are likely to reside closer to the app entry.

NLP Correlation. This clue states that if an app-entry and its invoked API are strongly correlated from a natural language perspective, we can infer that the API contributes to the app-entry to a large extent.

Given the complex nature of API call chains – i.e., an entry *ep* calls *api* via a chain of callers, Bluebird estimates the contribution of *api* to *ep* while factoring in the contribution of each caller along the chain to *ep*. The contribution estimation further penalizes callers deep in the call chain since those are likely to contribute less significantly to the top entry, unless there is a strong naming similarity.

Concretely, Bluebird estimates the contribution of a callee to a caller via cosine-similarity. It collects various textual identifiers and properties pertaining to the app entry (e.g., component and permission names, UI textual elements) and invoked methods (e.g., class, name, and non-primitive argument types). It then uses a transformer-based encoder to produce an embedding vector for the collected information. The contribution of a method to an entry point is estimated by the cosine-similarity of the produced embedding.

The following Equation details the contribution estimation.

$$Contribution(ep, api) = Max(Sim(ep, api), \\ Contribution(ep, c(api)) * Sim(c(api), api))$$

Where $c(api)$ is the direct caller of *api*, and $Sim(c_{i-1}, c_i)$ reflects the cosine-similarity of the embedding of a caller and a callee.

5.4.4 App Analysis: Extracting Context Clues

Our idea here is to propagate the sensitivity of an app entry to the APIs it invokes while considering contextual properties. Rules 9- 12 depict the program artifacts that we rely

on to generate the context constraints.

Path Sensitivity. Android APIs often implement conditional security specifications, meaning that a required access control may differ depending on supplied parameters and execution context.

```
1 public void incrementOperationCount(int uid, ...) {
2    if (Binder.getCallingUid() == uid // Path 1
3       || mContext.checkCallingPermission (MODIFY_NETWORK_ACCOUNTING, ..) == 1
              // Path 2
4    ) {  //privileged operation
```

Listing 5.4: API with path-sensitive Access Control

We note that this conditional nature complicates our auditing task – without reasoning about the execution context, we might incorrectly propagate the app's specification to called APIs. For example, consider the simplified access control implementation of the AOSP API `NetworkStatsService.incrementOperationCount`, shown in Listing 5.4. The API features two distinct access control paths. The first path requires the supplied *uid* to be equivalent to the calling app *uid*, while the second one requires the calling app to hold a specific permission.

Now, consider the scenario where the API is invoked in an app entry with `Process.myUid()` as its first parameter, but with no other explicit access control in that app entry. It would be incorrect to apply Rule 2 because the particular choice of the first parameter in the app entry is equivalent to the access control check in line 2 of the API implementation (Listing 2).

A straightforward solution to this problem is to interpret the presence of certain parameter values to an API call in an app entry as an implicit security specification. But this can lead to false positives if the parameter is not used in an access control check in the API implementation.

To solve the issue, we introduce Rule 12 that strikes a balance between lowering false positives and properly propagating a (potentially sensitive) app specification to the API. The corresponding constraints state the following: if the invocation context of an API *api* in an app entry *ep* matches specific arguments (i.e., those often used in path-sensitive access control implementations), and *ep*'s sensitivity is low, then we propagate *ep*'s sensitivity to *api* with low-confidence (i.e., *api* is less likely to follow *ep*'s sensitivity.)

Data Sensitivity. App entries invoking *data-sensitive* APIs are likely to impose a protection dependent on data flows. Unfortunately, due to the lack of a consensus or a guidance of

what constitutes a *data-sensitive* API, particularly in light of customization, it is challenging to identify whether an API is data-sensitive, at least without extensive framework analysis[8].

Bluebird addresses this challenge conservatively by relying on data flow tracking for all APIs. Specifically, Rule 9, and 10, state if an app provides a direct channel for a third-party entity to read (or update) the values returned (or set) by a sensitive API, the app should likely provide a protection. As such, the corresponding *full execution* constraints encode the intuition that we can propagate the app entry's specification to the API with a high confidence.

The reverse reasoning is not necessarily true. If the app provides a limited execution context, we cannot confidently propagate its protection to the APIs. Consider the case where a *source* API's returned value is not returned to the caller, the app is unlikely to enforce a protection.

The *limited execution* constraint encodes this uncertain intuition. It states that if an app entry provides some but not all access to the API's functionality (e.g., arguments reflects constant parameters, or app-specific resources that are not modifiable by the calling entity), we cannot faithfully trust the app-inferred specification. This constraint is conservative as the low confidence propagation only impacts low-sensitivity app entries.

Bluebird relies on data flow tracking to generate the proper execution constraint for an API's invocation site. Specifically, we construct and traverse the call graph from the app entry point to a reachable API. We then perform a backward inter-procedural data flow analysis from the API to resolve the arguments that flow to it. For each argument, we inspect the resolved values. If the value resolves to a method invoke statement, we check if it matches an *external source channel*[9]; i.e., a method that reads input from another application (e.g., reading content for a received Intent) or from the user (e.g., reading content from UI blocks). If so, we tag the argument as modifiable, and vice-versa. We also consider intricate cases where input implicitly flows from a few UI channels, such as Check Box, Slider, Toggles Button.

5.4.5 Probabilistic Constraint Solving

The inference constraints can be represented as probabilistic functions over the random variables involved. In practice, the computation of posterior marginal probability over all

[8] SuSi[10]'s classification is not fully applicable as it is mostly concerned with detecting sinks from the perspective of a remote adversary.

[9] We compiled the list manually.

these constraints is expensive, particularly in the cases of a large number of constraints. In our implementation, we leverage a graphical model, factor graph [78], to represent all probabilistic functions and allow an efficient computation. We use a standard off-the-shelf algorithm to compute posterior probabilities for the factor graphs. Our implementation is based on ProbLog [79], an off-the-shell factor graph engine (The details are hence elided).

5.4.6 Interpreting Auditing Results.

Bluebird outputs a posterior marginal probability for each API denoting its final sensitivity level as implied by all app-side security evidences and clues. The auditing procedure classifies the posterior probability as either `LOW` or `HIGH` by comparing it against a preset threshold (empirically determined).

Bluebird can lead to the three possible outcomes listed in Table 5.2. Category 3 reflects access control gaps, that is cases that indicate potential security issues and thus warrant further investigation. (Note that these outcomes do not reflect the obvious scenario where our audit cannot intrinsically work; i.e., where there are no apps invoking an API. We report the percentage of such cases in Subsection 5.5.2).

Table 5.2: Interpreting auditing results

Category	Initial Sensitivity	Final Sensitivity	Interpretation
1	**HIGH**	**-**	Do not Audit: No security issue
2	**LOW**	**LOW**	Compliance: No security issue
3	**LOW**	**HIGH**	Access Control Gap: Likely security issue

5.5 Evaluation

We implemented a prototype of Bluebird[10], which comprises three modules as illustrated in Figure 5.2. The modules are responsible for: (1) static framework analysis, (2) static/ UI app analysis, and (3) probabilistic inference. The static analysis modules are built on

[10]We will make the source code for research use available on request.

top of WALA [103] and analyze framework / preloaded app bytecode to construct analysis rules (priors and constraints) required for the inference. The probabilistic inference module fuses and solves the constraints.

Our prototype uses an offline NLP analysis subsystem, that works independently of the probabilistic analysis pipeline. The subsystem performs static app analysis, model construction and training.

The analysis is performed on a server equipped with a 32-cores CPU (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz) and 128G main memory.

5.5.1 Test ROMs

To test our Bluebird prototype, we collected a total of 14 ROMs, from public online repositories[59, 42] and from available physical devices. The samples include both AOSP and custom Android ROMs and span six versions, Android 8.1, 9, 10, 11, 12, and 13. They are customized by six vendors, at both ends of the customization spectrum.

We found that about 18% of framework files and 22% preloaded apps could not be properly decompiled by existing preprocessing tools or analyzed by WALA. Among those that could be analyzed, Bluebird reports the findings shown in Table 5.3. As shown, all vendors introduce new private APIs (column #3); Samsung, Xiaomi, and Vivo introduce more than 2000 APIs per ROM (more than 85% increase from AOSP) while, Amazon and ZTE introduce less than 430 APIs. The number of preloaded apps follows a similar trend.

5.5.2 Availability of App-Side Security Specifications

Our proposed approach hinges on the availability of invocation sites of APIs in apps. Besides, given the uncertainties involved in (1) connecting APIs to the app-side security specifications, and in (2) even possible contradictions in inferred API security specifications, our approach would work best when *multiple* invocations sites are found for an API, as app-side security specifications would lead to substantial uncertainty when only a small number of invocations are involved. We thus report the number of invoked public APIs, invoked private APIs, and the average number of invocation sites per API in preloaded apps, in Columns 5, 6 and 8, respectively. Bluebird specifically targets private vendor APIs; nonetheless, we report the results for public APIs for comparison.

The number of invoked private APIs varies across ROMs, ranging from as high as 43% in Samsung to as low as 12% in Oppo. Besides, the average number of invocation sites varies

Table 5.3: Statistics of the Android ROMs

(a) Apps and APIs statistics (Columns 1–6).

OS Image	**(2)** # **APIs**	**(3)** # **Private APIs**	**(4)** # **Preloaded Apps**	**(5)** # **APIs with app invocations**	**(6)** # **Private APIs with invocations**
Nexus 6P (V 8.1)	987	–	94	234	–
Pixel 2 (V 9)	738	–	127	195	–
Pixel 3XL (V 9)	738	–	129	196	–
Pixel (V 10)	2892	–	72	238	–
Pixel 6 (V 11)	1728	–	113	331	–
Pixel 5 (V 12)	1728	–	81	243	–
Pixel 6 Pro (V 13)	943	–	82	228	–
Fire HD (V 10)	1396	422	194	415	108
SM-A3058 (V 8.1)	5440	2551	268	1694	1118
Oppo (V 12)	2030	1011	113	259	124
Vivo (V 12)	3529	2201	145	729	372
Xiaomi Poco C3 (V 10)	3680	789	179	566	123
SM-N770F (V 13)	1820	635	325	548	141
Nubia Z50 Ultra (V 13)	1214	290	233	304	29

(b) Gaps, Invocations, and rule statistics (Columns 7–11).

OS Image	**(7)** # **Gaps*** **(TP% | FP%)****	**(8)** # **API invocations in apps**	**(9)** # **Avg invocations per API**	**(10)** # **Rules generated**	**(11)** # **Avg rules per API**
Nexus 6P (V 8.1)	19	840925	3593	1800401	1824
Pixel 2 (V 9)	23	158667	813	306242	414
Pixel 3XL (V 9)	22	167028	852	323577	438
Pixel (V 10)	19	46344	194	109270	37
Pixel 6 (V 11)	25	91856	277	276750	160
Pixel 5 (V 12)	22	300695	1237	658502	381
Pixel 6 Pro (V 13)	20	269115	1180	567064	601
Fire HD (V 10)	29 (83 | 17)	22499183	54214	54823476	39271
SM-A3058 (V 8.1)	168 (84 | 16)	3499350	2065	8330940	1531
Oppo (V 12)	21 (81 | 19)	10876122	41992	39849947	19630
Vivo (V 12)	104 (80 | 20)	594629	815	1546551	438
Xiaomi Poco C3 (V 10)	26 (74 | 26)	4410082	16394	13347222	3626
SM-N770F (V 13)	30 (80 | 20)	1938612	3537	6926108	3805
Nubia Z50 Ultra (V 13)	13 (70 | 30)	2640546	14508	10657652	8778

**We do not verify whether the gaps in AOSP are indeed false positives, since per our assumption, AOSP serves as the ground truth. For custom ROMs, the #gaps is reported only for private APIs.*
***The TP and FP percent for custom ROMs are estimated based on manual inspection of 10% of total gaps.*

significantly across APIs: particularly, *getter* APIs are more prevalent (e.g., Samsung's `ISemPersonaManager.getProfiles` has 230758 invocation sites).

5.5.3 Evaluating Auditing Results on AOSP

For a perfectly secure ROM – where APIs are protected consistently across the app and framework layers, Bluebird should report results belonging to category 2, as long as there are sufficient evidences and clues[11]. Hence, we rely on this intuition to gauge the accuracy of our tool. As in prior work [35], we assume that AOSP is perfectly secure, and use it to evaluate Bluebird. Column 7, Table 5.3 reports the number of public APIs that Bluebird identified as having access control gaps in AOSP. As shown, Bluebird reports a gap for 8% to 12% APIs which are by our assumption false positives. This means that *the remaining public APIs (88% to 92%) were found to be consistent with their app-side security specifications*.

We investigated all reported FPs manually – two authors were involved. The investigation identified that FPs were caused by the following: (1) limitation of our access control extraction logic in the framework-side (i.e., the API uses a difficult to detect logic); e.g., APIs using global state variables control dependent on access control checks, (2) limitations of app-side access control logic due to disconnections in the ICFG; e.g., apps using asynchronous callback mechanisms like live data which our implementation did not account for, (3) insufficient app-side evidences; e.g., API being invoked only from a few over-protective system apps, and (4) apps being overprotective; e.g., app-side sensitivity indicators meant for the app-specific use case but not generally applicable to the API being invoked. We did not attempt to verify whether they are indeed FPs since we consider AOSP as ground truth.

Observe that quantifying false negatives (FNs) is challenging due to the lack of ground truth. We resort to approximating FNs by constructing a *synthetic* vulnerable APIs dataset – where we maneuver Bluebird to assign a `LOW` prior to AOSP APIs that implement strong access control (i.e., we intentionally change the prior from `HIGH` to `LOW`). We then run Bluebird and count the portion of cases where it reports a `LOW` posterior, indicating an FN. On average, our analysis yields 17% FN rate. We manually investigated the cases and found that they are mostly due to: (1) An inferred `LOW` indicator for a sensitive UI blocks, and (2) insufficient app-side evidences.

[11] Category 1 APIs are excluded as they implement strong access control.

5.5.4 Landscape of Gaps in Custom ROMs

We then run Bluebird on custom Android ROMs. Column 7 in Table 5.3 reports the number of private APIs with access control gaps. As shown, the number of gaps ranges from 15% to 44%. Due to the lack of ground truth, we estimate false positives in the reported gaps through manual investigation. Two authors examined 10% of the gaps and an FP is reported if both agreed. The authors rely on domain knowledge to verify whether a gap is an FP. Examples of manually detected FPs are `WifiManager.getWifiApInterfaceName()` and `KnoxCustomManager.getVibrationIntensity`. The FPs were caused by the same reasons as in AOSP analysis.

5.5.5 Impact of the Preset Probabilities

Bluebird uses three preset probabilities: $p_h = 0.9$, $p_l = 0.1$, and $p_m = 0.8$. We measure the impact of changes in these probability constants by calculating the percentage of the auditing categories using the following four experiments:

EXP 1. $p_l = 0.35$ and $p_h = 0.8$ in Rules [2] and [3].

EXP 2. $p_h = 0.8$ in Rules [7] and [8].

EXP 3. $p_l = 0.2$ and $p_h = 0.8$ in Rule [10].

EXP 4. $p_m = 0.6$ in Rule [11].

All experiments are performed in isolation (i.e., probabilities not mentioned in the experiment remain unchanged as in Table 5.1) and on the same ROM (Pixel 6 Android 11) to assess the impact of each change precisely and consistently.

Table 5.4: Impact of probability values

Category	EXP1	EXP2	EXP3	EXP4
1	234	234	234	234
2	72	74	75	74
3	24	22	21	22

Table 5.4 shows the number of APIs reported by Bluebird for each category using the four experimental settings. As shown, the impact of the probability values is largely minimal. We

manually investigated the few API instances that did exhibit some change, and concluded it mainly occurred due to insufficient app-side clues/evidences – i.e., the preset probabilities may have higher impacts in cases where there are fewer hints.

5.5.6 Comparison with Convergence-Based Inconsistency Analysis

A closely-related line of research to our approach is convergence-based inconsistency analysis, which aims to identify access control flaws by comparing security specifications leading to similar resources. The works perform convergence analysis either within the framework layer [64, 1, 111] or across different layers of the Android stack [73, 152]. Bluebird is conceptually a cross-layer access control analysis framework, as it conducts the auditing procedure by learning security specifications from the app layer.

To qualitatively demonstrate the benefits of using Bluebird over existing convergence-based work, we investigate a subset of reported access control gaps. Particularly, we intentionally select those leading to actual vulnerabilities (Please refer to Section 5.6 for more details about the cases) and check whether prior works are able to detect them. Our analysis confirms that out of the 11 reported vulnerabilities, 7 *cannot be caught by existing convergence-based techniques*, as (1) the vulnerable APIs do not share a convergence point with other framework APIs, and (2) the vulnerable APIs do not contain any cross-layer access (e.g., no file access, and no native method invocation). A prominent example of these vulnerabilities is the case discussed in Section 5.2.

While this finding clearly demonstrates its complementary nature, Bluebird is inherently more coarse-grained compared to existing approaches. It can only indicate the presence of a gap (i.e., an access control is absent), while other work can identify precisely the missing access control (e.g., exact permission). As Bluebird relies on app-side sensitivity indicators, including natural language text from UIs, it cannot map those to concrete traditional access control checks.

5.5.7 Comparison with Poirot

Our work is inspired by Poirot [35], an Android access control analysis and recommendation tool, that similarly leverages probabilistic inference to account for inherent access control-related uncertainty. Contrary to our work, Poirot is specifically tailored towards *framework-level* access control analysis and works *exclusively by relying on framework-level* security-relevant hints. Bluebird is a parallel probabilistic security analysis framework for modeling of evaluating access control *across the app and framework layers*.

Here, we demonstrate the fundamental differences with Poirot. We obtained a list of access control inconsistencies reported by Poirot for Amazon Fire HD 10 and compared it against the results we obtained using Bluebird (refer to Row 8 in Table 5.3). Out of the 24 gaps reported by Bluebird, Poirot was able to identify three, hence missing 21. Besides, out of the 12 true positives reported by Poirot, Bluebird was only able to identify three. We present a sample case-study from each category with a brief root cause discussion to showcase the complementary nature of Bluebird and Poirot.

Poirot-Exclusive: Our manual investigation shows that Bluebird misses these cases due to the lack of invocation sites in preloaded apps – hence, the app-side evidences and clues that Bluebird relies on, are absent. An example of such cases is `AmazonInput.setInputFilter`, reported by Poirot, which is not invoked by any preloaded apps.

Bluebird-Exclusive: We concluded through manual analysis that Poirot misses these cases because the APIs lack comparable counterparts in AOSP or other parts of the custom framework. Thus, Poirot was not able to generate significant security-relevant hints to connect the target APIs' resources to other framework-level resources. For instance, `SmartSuspendManagerService.setScheduleType`, reported exclusively by Bluebird, accesses resources that exhibit no semantic or structural relations with other resources in the framework.

5.6 Case Studies of Access Control Gaps

Not all reported APIs with access control gaps are exploitable. Nonetheless, they do warrant further investigation as Bluebird probablistically detects the gaps based on *dominant* app-side hints. To demonstrate that some gaps are indeed exploitable, we selected a few whose logic is comprehensible (some reports occur in custom proprietary functionality) and built end-to-end PoCs. Table 5.5 reports our manually confirmed vulnerabilities; which can allow third-party apps to alter various system settings and mount DoS attacks without permission. We reported the findings to the vendors. All reports have been acknowledged, and nine have been fixed. Next, we describe two selected cases. (Refer to Appendix for a discussion of three other cases).

Table 5.5: Summary of discovered APIs with access control flaws

OS Image	Service API	Security Implication	Vendor Response
Fire HD 10	SmartSuspendManagerService .setScheduleType	Modify Use Specified Setting	Acknowledged*
Fire HD 10	AmazonAspService .command	Modify critical Audio related Settings	Acknowledged*
SM-A3058	PersonaPolicyManagerService .setSecureFolderPolicy	Modify Secure Folder (provided by Knox) Policy	Acknowledged*
SM-A3058	SemClipboardService .showDialog	DoS Attack	Won't Fix
SM-A3058	SemClipboardService .dismissDialog	DoS Attack	Won't Fix
SM-A3058	IWifiService .setWifiSharingEnabled	Change wifi settings	Acknowledged
SM-A3058	IBluetoothManager .shutdown	Shutdown Bluetooth connections	Acknowledged
SM-A3058	IAccessibilityManager .semOpenDeviceOptions	Interface Illusion	Acknowledged
SM-A3058	IAccessibilityManager .OnStartGestureWakeup	Change accessibility Settings	Acknowledged*
Fire HD 10	AmazonPowerManager .setBatteryChargingVoltage	Alter the power voltage	Acknowledged*
SM-A3058	PersonaManagerService .setAppSeparationDefaultPolicy	Manipulate app separation policy	Acknowledged*

*The vendor awarded a bounty.

Manipulating Samsung's app security policies. Bluebird reported a gap in `PersonaManagerService.setAppSeparationDefaultPolicy`, a private API in Samsung, that allows to reset app-specific *separation policies*. As policy updates can only be initiated by device administrators on Samsung, the API is clearly sensitive. However, we found that the API enforces no access control. Bluebird was able to identify the gap thanks to various UI-based

evidences and clues in a preloaded app *SecAppSeparation*. We explain two examples next.

Bluebird identified that the API is invoked in the Activity `ProvisioningActivity`, among other app entries. The Activity's UI includes textual elements that our NLP model predicted to be sensitive with a high confidence (>0.99). Examples of extracted text include "This separation can allow your *IT admin* to provide *different security restrictions*" and "*Keep in mind* that apps in this folder are still managed by your *IT admin*".

Bluebird leveraged a similar UI clue to identify that the API `setAppSeparationDefaultPolicy` is relevant to the inferred sensitivity. Particularly, it found the following sentences "Configuring apps for separation" and "Your IT admin has *separated apps* in this folder" which share a high naming similarity with the target API, and thus result in a contribution score (∼0.8) for this API to the sensitive Activity.

By combining these UI-based hints, Bluebird concludes that the API is sensitive with a high posterior probability (∼0.95).

Manipulating critical audio related settings. Bluebird reported a gap in the API `AmazonAspService.command` defined by Amazon in Fire HD 10. Bluebird found numerous invocations of the API, spanning a few preloaded apps such as *Alexa* and *amazon.speech*. While the API enforced no access control (thus, a `LOW` prior), all its app-side invocations occurred in highly-sensitive services, thus leading to a high posterior probability. We manually investigated the API's implementation and discovered that it allows manipulating many critical settings such as speaker volume, microphone mute status, battery voltage, voice messaging status, speaker volume decibel values and others.

5.7 Threats to Validity and Limitations

We discuss here various factors that affect the significance of the Bluebird approach along with its limitations.

Bluebird is dependent on the availability of app-side invocations of private APIs, and the availability of (detectable) sensitivity indicators, which poses a threat to the validity of our findings in Section 5.5. The number of private APIs' invocations in the preloaded apps, analyzed in our collected ROMs corpus, may not be representative of other custom ROMs. Existing research [34] reports that vendors may introduce *Residual* private APIs that are not invoked by any preloaded apps. In such cases, Bluebird is inherently limited. Besides, Bluebird is dependent on the accuracy of app-side sensitivity indicators extraction. We mitigate this threat by leveraging various methods, each tailored to identify a different indicator.

Another threat to validity lies in heavily obfuscated preloaded apps, which may hinder both the identification of private APIs' invocation sites as well as their corresponding sensitivity indicators, particularly those enforced in code.

Last, Bluebird assumes that app-side security specifications are always relevant to invoked APIs. However, this assumption may not hold if apps are over-protective. We attempted to mitigate this threat by accounting for uncertainty via probabilistic inference.

Chapter 6

Analyzing Data-Driven Customizations in Android

This chapter is adapted from the paper [127] that introduces Ariadne, a static-analysis technique for detecting access control inconsistencies arising from data-driven framework customizations. Ariadne focuses on *framework-level data structures* and their implicit security implications. It introduces a novel abstraction, the *Access Control Dependency Graph*, together with a selective flooding algorithm that propagates requirements while accounting for uncertainty.

6.1 Introduction

A particular type of customization that vendors perform at the framework-level in Android is *data-driven*, occurring when vendors introduce or alter *data holders*, which are framework variables accessible through public APIs. Depending on the sensitivity of stored content, data holders are subject to access control requirement (e.g., permissions), enforced in an API's implementation.

Data-driven customization risks introducing vulnerabilities like access control anomalies (e.g., an under-protected new data holder, or a data holder with inconsistent access control across different APIs), which malicious app developers could exploit. The security implications of data-driven customization, however, are largely unexplored. Data-driven inconsistencies are challenging to detect/mitigate using existing tools, as they exhibit unique patterns not tackled by prior work [64, 1, 111, 73, 152, 128].

Data driven vendor customization comes in different forms: (1) Introduction of new data holders. (e.g., Samsung introduces a new data holder in `TelephonyManager` called `semPhoneInternal` to save the state of newly defined custom telephony properties). (2) Modifications to existing AOSP data holders, e.g., by adding a new field to an AOSP data holder type, or adding a new data holder type (with additional fields) that extends an AOSP data holder type (e.g., Samsung adds new fields to `ICCRecord`, a data holder in AOSP's `TelephonyManager` to save the Samsung-defined properties added to `ICCRecords`). (3) Introduction of methods accessing/modifying custom data holders, with different granularities and implications, e.g., a vendor may introduce a utility method to read all entries in a custom Java Collection instance (e.g., Xiaomi defines a utility method `getUserRestrictions` to read all entries of a custom data holder `userRestrictions`), another for accessing a single entry in the instance (e.g., Xiaomi defines another API `getDeviceOwnerUserRestrictions` to access only the device owner's `userRestrictions`, , and a third for inspecting if a given entry exists in the instance (e.g., Xiaomi defines a third API `hasExtention` to check whether an entry exists in `mExtensions`, a list of `UserRestriction`).

In our study of 11 custom ROMs, we found that data-driven customization is highly prevalent: e.g., 45027 instances found in Samsung ZFlip v13 (Section 6.6). Their complexity and *implicit access control implications* make them potential attacker targets.

Implicit Security Implications. Framework data holder instances can be accessed/modified either by system processes directly, or by apps using APIs. Depending on the sensitivity of the data, the APIs may enforce access control (e.g., writing an entry to *System.Settings* provider requires `WRITE_SECURE_SETTINGS` permission). Data-driven customization should ensure that new/existing data holders are consistently protected by related APIs. This is hard to accomplish due to the limited, often non-existent, security specifications.

Data holders have two kinds of implicit "*security specifications*"[1] that may be overlooked during customization: those implied by (i) Java objects semantics (e.g., an access control-protected field implies that the containing class object should also be protected - e.g., `TelephonyManager` has a data holder `mIccRecords` of type `IccRecords`. An access control protected field `mImei` of class `IccRecords` implies that the `mIccRecords` object should also be protected), or (ii) supported operation semantics (e.g., access control for an operation that checks if an entry exists in a collection implies that reading the entry should also be protected - if `hasIccCard` from `TelephonyManager` is protected, then `getIccCards` should also be protected). These implications can be non-deterministic (e.g., an access

[1]Following [128], we use "security specification" in a broad sense, to refer to the security policy inferable from language and data types semantics.

control-protected class object may or may not imply that one of its member fields should be protected). To effectively utilize non-deterministic implications, we must model their associated *uncertainty* accurately.

Prior inconsistency detection approaches in Android either 1) rely on convergence [112, 64, 1] or 2) utilize uncertain *hints* such as control-dependency, naming similarity, or UI based security indicators [35, 128]. However, none of them can reason about implicit security implications in data holders and hence cannot accurately catch underlying data-driven inconsistencies. The diversity in data holders necessitates modeling of their structure, internal fields and relations among them, involving unique challenges (Section 6.2) that require dedicated analysis: (1) converting the diverse data types of framework data holders into a cohesive, unified abstraction for effective analysis, and (2) using ambiguous access control implications connecting data holders. Existing tools do not support such modeling. We present a new tool, Ariadne[2], which addresses these challenges as follows:

Novel Abstract Representation. To tackle the complexity and diversity of data holder types, Ariadne transforms type information of variables into a novel unified abstraction, the *AC dependency graph*, which represents data holders as nodes, relations connecting them using edges, and access control implications connecting nodes via labels to the corresponding (directed) edges. The edges are either *deterministic* or *non-deterministic*. For example, an edge connecting a parent class to its child node (representing a member field) is labeled as non-deterministic (since a child node does not necessarily inherit its parent access control enforcement).

Access Control Inference and Inconsistency Detection. Ariadne leverages the statically extracted access control information and deterministic edges to *annotate* nodes with required access control. The generated graph is often partially annotated. To predict missing annotations, Ariadne uses the non-deterministic edges. Specifically, it treats the graph as a flow network [104] and uses flooding [5] to transitively propagate and predict access control annotations along the labeled edges (Subsection 6.5.2). Finally, data-driven inconsistencies are detected when the predicted access control annotations are stronger than the implemented ones.

Results. With Ariadne, we performed the *first systematic study* of data-driven customization in Android using two AOSP and 11 vendor-customized ROMs. On AOSP, Ariadne predicts correct access control in 553 (97%) of data-driven related APIs. On custom ROMs, from seven vendors, it detects 90 unique data-driven inconsistencies, with a manually estimated false-positive (FP) rate of 11.8%. To demonstrate the impact of the inconsistencies, we built end-to-end exploits for 13 cases with various consequences; including enabling/disabling

[2]In Greek mythology, Ariadne was a Cretan princess who could navigate mazes and labyrinths.

arbitrary critical packages in Tecno, launching random Activities with system privilege in Vivo (an unauthorized app can trigger system-protected Activities such as *factory resetting the device*) and altering critical data policies in Samsung.

Ariadne detects vulnerabilities that existing tools cannot, constituting an important new addition to the suite of Android inconsistency detection tools *complementing* (not replacing) existing tools to provide more comprehensive protection.

We summarize our contributions as follows: We

- identify the hitherto understudied problem of access control inconsistencies in the Android framework resulting from *data-driven customization*, (Section 6.2)
- propose Ariadne. (Section 6.3,Section 6.4,Section 6.5), a novel approach we use for the *first systematic study* of Android data-driven customizations, showing Ariadne's effectiveness on 11 custom Android ROMs, discovering 90 unique access control inconsistencies, (Section 6.6) and
- show how Ariadne complements existing techniques by detecting 30 unique new inconsistencies, not caught by other tools (ACMiner, Poirot and Bluebird). (Section 6.7)

6.2 Motivation

Prior work has proposed detecting access control flaws in the Android framework using a range of techniques such as convergence-based inconsistency detection as in Kratos [111], IAceFinder [152] and probabilistic inference as in Poirot [35] and Bluebird [128]. However, these are limited, if not incapable, in their ability to catch *data-driven inconsistencies*.

This lapse stems primarily from not considering implicit security specifications implied by: (1) Java object semantics and (2) semantics of Android APIs operating on data holders (e.g., an operation requiring access control may imply that other related operations require similar access control). Next, we elaborate on this insight.

6.2.1 Java Objects Access Control Semantics.

The Android framework is implemented using the object oriented programming paradigm. To detect data driven inconsistencies, it is important to understand access control implications of various relations among classes and their member fields.

Example. To illustrate this insight, we use `SemTelephonyController`, a custom system service defined by Samsung to tailor AOSP's Telephony framework. Figure 6.1 shows

its UML class diagram, simplified by omitting some data types and methods for clarity. `SemTelephonyController` uses a number of AOSP data types (shown in green) and customizes some (shown in red). The customization entails adding new (i) fields like `mEuimid` and (ii) utility methods like `getEuimid`.

Implicit access control semantics. To evaluate the correctness of access control, it is essential to consider relations among data types (e.g., a field is a member of a class instance) and understand their relevance to access control propagation.

Specifically, if an API enforces a permission to protect access to a variable, then the permission requirement may need to be propagated to other relevant variables. In the example, the custom API `isSupportLteCapaOptionC` enforces access control to protect reading the field `mVendorConfigState.mSupportLteCapaOptionC`. This requirement should automatically flow to the containing class' instance `mVendorConfigState`, because reading it inherently allows reading its member fields (via invoking `getVendorConfigurationState`). Hence, the API `getVendorConfigurationState` should require a similar access control as in `isSupportLteCapaOptionC`. This conclusion would go unnoticed unless the analysis properly handles access control implications across classes, fields, nested fields, etc.

In practice, extracting Java object access control semantics faces four major technical challenges due to (1) the need for precise data flow analysis, (2) the sheer number of (custom) objects in the Android framework, (3) the diversity of the underlying hierarchical structures, and (4) the uncertainty involved.

TC1: Precise data flow analysis. The analysis needs to be both *field-* and *objective-sensitive*.

Field-sensitivity is needed to precisely propagate access control up to class and its nested member fields, if any. If a class contains many member fields, the analysis should ensure propagating access control only to the appropriate field.

Objective-sensitivity is required to associate access control with the right *object* of a class. As various APIs may access different instances of the same class type, object-insensitive analysis can lead to overly restrictive access control inference. This may result in false positives – we illustrate this through a case study in Section 6.9.

TC2: Defining the analysis scope. Precise data flow analysis is expensive in terms of computational complexity and memory overhead, making it impractical due to the sheer number of objects needing analysis in custom Android codebases. To scale, the analysis must be selective, focusing only on customized framework variables likely to be data holders.

TC3: Diverse data types. Data holders may follow diverse hierarchical structures. For example, classes can define nested classes, child classes can have their own set of fields while

still inheriting other fields from the parent class. To detect access control implications, the analysis should be able to (1) represent the complex hierarchical structure and (2) trackaccess control relations among elements.

TC4: Inherent Ambiguity. Some access control implications can be uncertain. For example, an access control requirement for a class instance does not necessarily imply that *all its fields* require the same access control, but rather *at least one of them does.*

6.2.2 Operation Access Control Semantics

APIs that operate on data types may reflect different operation semantics (e.g., update, set, get) and granularity (e.g., update all versus update some elements). These operation semantics and granularity can be used to carry access control implications to other related APIs. For example, if an API enforces an access control to verify the existence of *an element* in a list, but another API *reads all elements* in the list without enforcing any access control, then this is likely an inconsistency.

```
1  private HashMap<String, SecureSpacesExtension> mExtensions;
2  public List<String> getExtensions() {
3    return new ArrayList(this.mExtensions.keySet());
4  }
5
6  public boolean hasExtension(String extensionName) {
7    checkCallerIsSystem();
8    return this.mExtensions.containsKey(extensionName);
9  }
10
11 public class SecureSpacesExtension {
12   public ArrayList<UserRestriction> userRestrictions;
13 }
```

Listing 6.1: getExtensions and hasExtension APIs

Example. Xiaomi defines two APIs `getExtensions` and `hasExtension` in a custom system service, `SecureSpacesService` (Listing 6.1). The two APIs operate on `mExtensions`, a HashMap that maps a String key to a value of a custom composite data type *SecureSpaceExtension* (Line 11-Line 13). The APIs reflect different operations performed on the data holder `mExtensions`. `getExtensions` is a generic getter as it returns *all* keys from `mExtensions` (Line 3) whereas `hasExtension` checks for the existence of a key in `mExtensions` (Line 8). `hasExtension` enforces a system-level access control (Line 7)

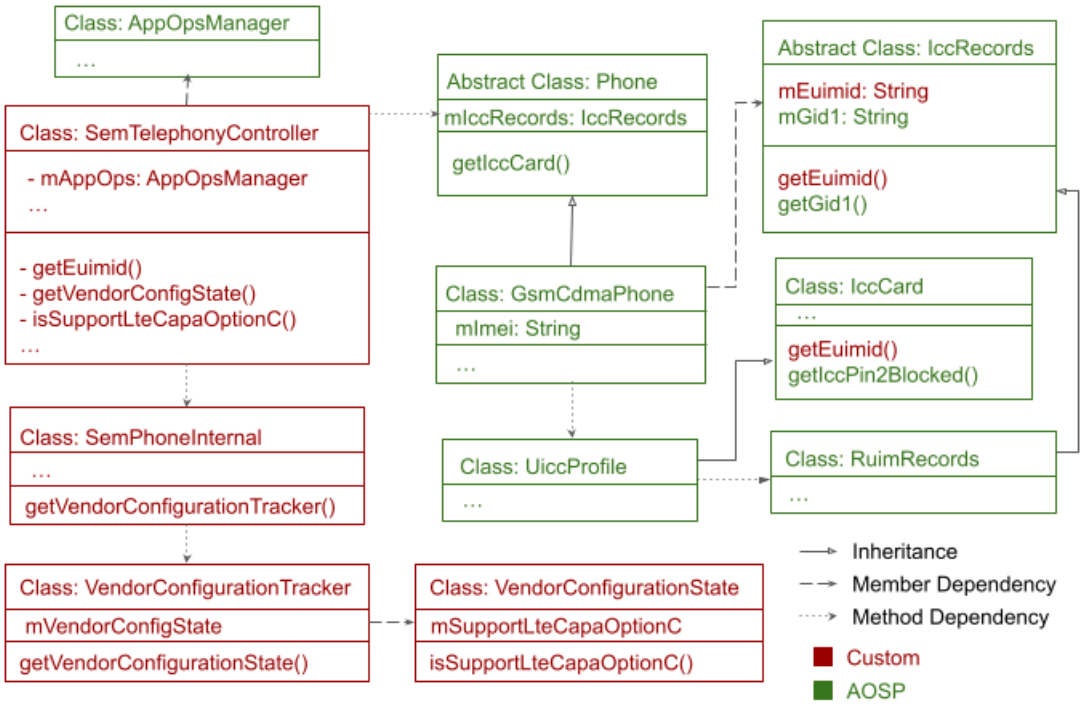

Figure 6.1: `SemTelephonyController` UML class diagram

The access control implemented at the *existence check* operation implies that the *get* operation should be similarly protected. However, as shown in the listing, this access control implication is not respected. The getter `getExtensions` does not enforce any access control – a likely vulnerability.

TC5: Modeling operation semantics. The analysis should account for operation semantics; otherwise, inconsistencies like the one in Listing 6.1 would go undetected. Besides, the analysis should infer the operation granularity, if any – which may require modeling developer-implemented logic in APIs.

6.3 Approach Overview

To address the challenges in Section 6.2, we propose Ariadne, a systematic approach for inferring access control inconsistencies caused by data-driven customization. Figure 6.2 shows Ariadne's two-stage workflow.

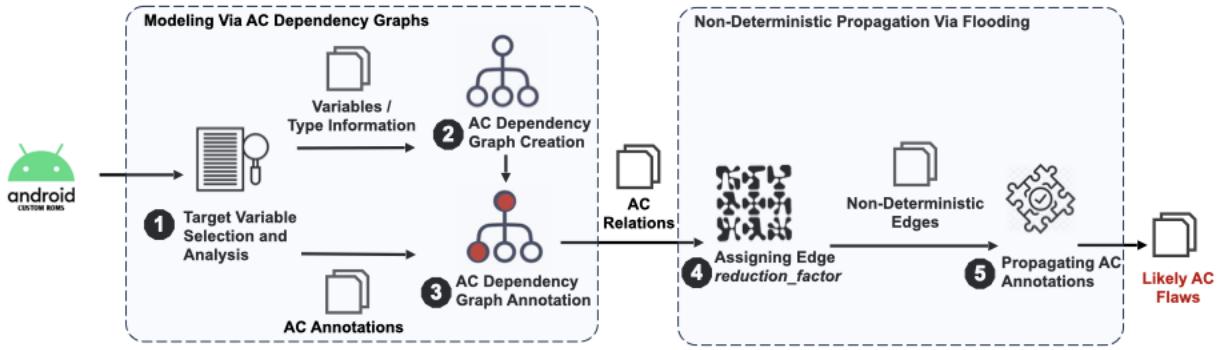

Figure 6.2: Ariadne system diagram

Stage 1: Modeling via AC Dependency Graphs. Given an Android ROM, Ariadne performs static analysis (Step 1) on defined system services[3] to identify reachable framework variables and corresponding type information. To address TC1 and TC2, Step 1 performs *selective field- and object-sensitive* analysis to recover a set of *target variables* that are potential data holders, chosen based on the criteria discussed in Subsection 6.4.1.

Step 1 also extracts *Access control annotations* for the data holders: fine-grained access control information, summarizing access control requirement for various operations (and granularity) performed on data holders (TC5). For example, an annotation might correspond to (i) a *normal permission* for *reading* a data holder, and (ii) a *system permission* to *write to* the same data holder.

Recovered data holders and type information, are then transformed (Step 2) into a new unified abstraction, the *access control dependency graph* (TC3). Nodes in the graph represent data holders. Labeled edges express the aforementioned access control implications, including Java objects relations and operation semantics, partitioned into: (1) *deterministic* and (2) *non-deterministic*; the former allows a direct propagation of access control implications between nodes while the latter requires further reasoning.

Step 3 annotates nodes in the access control dependency graph. The collected access control annotations may only cover those nodes reachable from protected APIs. Hence, Step 3 may generate a *partially-annotated* AC dependency graph. We predict missing access control annotations as follows: (1) Automatically propagate access control annotations between nodes connected via *deterministic* edges. (2) Use a custom flooding [5] algorithm that can - (i) allow quantifying the uncertainty, and (ii) conditionally propagate access

[3]Android system services define APIs that can be invoked by apps.

control annotations to the nodes connected via *non-deterministic edges*, as described next.

Stage 2: Non-deterministic Propagation via Flooding. Due to the uncertainty in propagating access control annotations via non-deterministic edges (TC4 (Section 6.2)), we propose a new graph-based technique to generate a set of possible access control annotations for nodes missing annotations. The annotations are *determined* and *ranked* based on the uncertainty associated with non-deterministic edges (transitively) linking the node to other nodes in the graph. Specifically, Ariadne treats the AC dependency graph as a flow network where nodes are categorized into sources and sinks that produce and consume access control annotations. Sources are nodes assigned a deterministic access control annotation in Stage 1, while sinks are those missing annotations. Edges denote pathways enabling a *controlled* propagation of the annotations from sources to sinks based on their uncertainty labels.

Similar to traditional flow networks, a node's access control annotation propagates to *all* its neighboring nodes except for the source node where the annotation originated. However, unlike traditional flooding algorithms which allow unconditional propagation, our setting requires a *conditional* propagation to model the uncertainty of the edges' access control implications. To address this, Ariadne implements a customized selective flooding algorithm that conditionally propagates access control annotations through non-deterministic edges.

To determine the set of possible annotations for sinks, Ariadne begins by assigning each access control annotation in the sources an initial weight, *relevance_weight*, denoting our initial belief of the access control annotation being *relevant* to the nodes. It then uses a flooding [119] algorithm to adjust the *relevance_weight* of the access control annotations based on connected nodes. The newly updated *relevance_weight* reflects an aggregated value computed based on the (number of, and types of) edges connecting the node to other nodes.

Ariadne assigns a *reduction_factor* to each edge in the AC dependency graph based on its initial labels, e.g., an edge connecting a parent class to a member field has a *reduction_factor* of 0.6 (meaning that the member field has a 60% chance to inherit its parent's access control annotations). The flooding algorithm then uses the *reduction_factor* for controlled propagation of new annotations, adjusting their *relevance_weight* along the way. When multiple incoming access control annotations flow into a sink, Ariadne combines their *relevance_weight* using the noisy OR [114] operation. We describe propagation further in Subsection 6.5.2.

Detecting Data-driven Inconsistencies. Finally, Ariadne propagates the ranked access control annotations of nodes to the corresponding APIs (i.e. those that operate on the node's variable). Inconsistencies are detected when the API enforces a weaker access control than Ariadne's recommendation.

6.4 Modeling via AC Dependency Graphs

We explain the details of Stage 1 and its individual steps (Section 6.3) in each subsection.

6.4.1 Step 1: Target Variable Selection

We want to analyze customized data holders. Since a typical custom framework defines a large number of variables, many of which are irrelevant to access control, we rely on the following selection criteria to narrow the scope, trying to identify *target framework variables* likely to be data holders. This allows scaling the analysis to large codebases (TC2).

- Local variables are unlikely to require access control. Thus, we only focus on global variables.
- Variables defined in internal framework classes typically do not require access control as they are not exposed to apps. We only focus on variables transitively reachable to apps via public APIs.
- Primitive variables not connected to others (e.g., not a member of class) cannot independently generate access control implications. Hence, we exclude them.

We find that these strategies lead to a substantial reduction of target variables. The reduced set contains variables belonging to two types; (1) *self-contained*, containing methods that operate only on member fields, like `SecureSpaceExtension` in Listing 6.2, and (2) *non self-contained*, like common Controller classes (e.g., `SemTelephonyController` in Figure 6.1).

While theoretically both types can be protected by access control, we only focus on the first. Analyzing non self-contained types, though more comprehensive, has two major disadvantages: (1) Access control inconsistencies in non self-contained types are unlikely to be caused by data customization (Subsection 6.6.2), and (2) it is more expensive as it generates highly-complex AC dependency graphs. By leaving them out, we risk false negatives, which are very unlikely as we show later (Subsection 6.6.2).

Ariadne collects **target variables** by extracting system services similar to prior work [12, 1, 64] and conducting reachability analysis to extract reachable classes; i.e., those instantiated or statically accessed by the service. Each identified class is then inspected to extract corresponding instance and static fields declared by the class or by any of its parent classes. Fields not conforming to the selection criteria above are excluded from the analysis.

Table 6.1: Relations between AC Dependency Graph nodes

Relation	Edge Propagation Type
Member to containing class	Deterministic
Class to member fields	Non-deterministic
Specific to generic	Non-deterministic
Generic to specific	Non-deterministic
Siblings	Non-deterministic

6.4.2 Step 2: Graph Creation

An AC dependency graph is a directed graph where nodes represent target variables and edges encode relations among them. The graph has the following three properties: it (1) allows expressing the variety and complexity of Android composite data types using a unique representation, (2) allows annotating access control requirements up to the granularity of class fields, and (3) encodes access control implications between them.

Graph Node denotes a target variable. It stores the variable name, data type, defining class, and instantiating class. We divide graph nodes into three categories: (1) *Simple* nodes encompass Java primitive types and certain Android-specific types such as `Bundle` and `Parcel`. They correspond to the leaf nodes in the graph because they cannot have member fields. (2) *Complex* nodes represent types with one or more member fields, hence they have one or more outgoing edge(s). (3) *Collection* nodes represent a unified type for storing and manipulating a group of Simple, Complex, or Collection types. In addition to Java types extending the `Collection` interface, we also consider `Maps` and `Arrays` as Collection types. Typically, `Collection` nodes should have an outgoing edge for each element in the collection; however, due to their dynamic nature, it is often difficult to resolve the number statically. Instead we create two special outgoing edges that connect a `Collection` node to two synthetic nodes, `ALL` and `SOME`, representing two possible member (access) granularities. `ALL` denotes the granularity where *all* members of the collection are accessed; whereas `SOME` represents accesses to a subset.

Directed Edges encode relations connecting pairs of nodes. Each edge label corresponds to one of the relations in Table 6.1.We group the relations into two classes, *deterministic* and *non-deterministic.* Deterministic relations reflect an *always true* access control implication – if two nodes are connected via a deterministic edge, their access control annotations should be equivalent. For example, an edge from a member field node to its containing class object node is deterministic, hence the field's access control annotation can be propagated to the containing object node.

A non-deterministic relation encodes uncertain access control implications from Java objects and API semantics. For example, edges that connect sibling nodes (i.e., fields contained in the same class) are non-deterministic, therefore, their access control annotations are not necessarily equivalent.

Algorithm 1: Construct an AC dependency graph

Input : $targetVars$
Output : $ACDepGraphNodes$
Requires :

1. $getNodeCat(var)$: Returns the node category for var.
2. $createNode(nodeCat, ID, type, struct)$: Creates a new node of category $nodeCat$.
3. $addEdge(n1, n2, edgeType)$: Connects ordered nodes $n1$ and $n2$ with an edge labeled $edgeType$.
4. $getInnerCollectionType(type)$: Returns the Java type of members of a Collection.
5. $getNodes(type, struct)$: Returns a set of all nodes with Java type $type$ and declaring class $struct$.
6. $getMembers(type)$: Returns a set of all member fields of class $type$.

```
1  ACDepGraphNodes=[]
2  for tVar ∈ targetVars do
3      ID = tVar.ID, type = tVar.type, struct = tVar.getDeclaringClass()
4      nodeCat = getNodeCat(tVar)
5      node = φ
6      switch nodeCat do
7          case SIMPLE do
8              node = createNode(SIMPLE, ID, type, struct)
9          end
10         case COMPLEX do
11             node = createNode(COMPLEX, ID, type, struct)
12             memberNodes = []
13             for m ∈ getMembers(type) do
14                 memberNode = createModel(m)
15                 addEdge(node, memberNode, containsMember)
16                 memberNodes.add(memberNode)
17             end
18             for (c1, c2) ∈ childNodes do
19                 addEdge(c1, c2, siblingOf)
20             end
21         end
22         case COLLECTION do
23             node = createNode(COLLECTION, ID, type, parent)
24             innerType = getInnerCollectionType(type)
25             someNode = createModel(innerType)
26             someNode.ID = SOME
27             allNode = createModel(innerType)
28             allNode.ID = ALL
29             addEdge(node, someNode, collectionOf)
30             addEdge(node, allNode, collectionOf)
31             addEdge(someNode, allNode, generic_specific)
32         end
33     end
34     for insNode ∈ getNodes(type, struct) do
35         addEdge(node, insNode, shareSuperType)
36     end
37     ACDepGraphNodes.add(node)
38 end
39 return ACDepGraphNodes
```

Graph Construction (Algorithm 1) is our process for constructing access control dependency graph(s) from a given Android system service. It takes the set of target variables *targetVars* generated in Step 1 (Subsection 6.4.1) and their resolved concrete types, and

produces the corresponding AC dependency graphs.

To resolve the concrete types of variables, we track each new object-allocation site (e.g., through identifying allocation instructions such as `SSANewInstruction`). Different instances of the same type will each have their own nodes. Objects extending the same super-class will be similarly allocated dedicated nodes. This way, the AC dependency graph ensures that tracking occurs at object-instance level (nodes represent unique object instances). We presented a case study in Subsection 6.2.1 (Listing 6.3) to illustrate the importance of object-sensitivity.

We define a number of helper functions (omitting implementation details for brevity). For example, function *getNodeCat* evaluates the Java type of a target variable and classifies it into one of the categories *simple*, *complex*, or *collection*.

For each target variable $tVar \in targetVars$, the algorithm initializes its AC dependency graph by creating a node whose type reflects *tVar*'s category (line 8, line 23, and line 11). It then adds nodes to the graph as follows:

- If the node *tVar* is a *complex* type, obtain its member fields and create corresponding nodes. For each pair of ordered nodes <*struct, member*>, add a *containsMember* edge to abstract the relation (line 15). Similarly, for each pair <*member, member*>, add a *siblingOf* edge.
- If the node is a *collection*, create two special nodes `ALL` and `SOME`. Assign the nodes the same data type as the inner type of the collection. Next, create two special edges, labeled *collectionOf*, to connect the ordered nodes pair <*parent,* `ALL`> and <*parent,* `SOME`>(line 29-line 30). Similar to the *siblingOf* edge add a *generic-specific* edge to connect the pair nodes <`ALL`, `SOME`> (line 31).

In line 35, the algorithm connects two AC dependency graphs. This happens when a node is found to be connected to a node belonging to an already constructed graph (e.g., when two APIs in a system service operate on two different members of the same target variable. As shown, the algorithm constructs the graph recursively repeating these steps for each member type (line 15, line 25, line 27).

```
1  private HashMap<String, SecureSpacesExtension> mExtensions;
2  public List<String> getUserRestrictions() {
3    ArrayList<String> restrictions = new ArrayList<>();
4    for (SecureSpacesExtension extention : this.mExtensions.values()) {
5      for (UserRestriction restriction : extention.userRestrictions) {
6        restrictions.add(restriction.name);
7        ...
8    return restrictions;
9  }
10 public List<String> getDeviceOwnerUserRestrictions() {
11   checkCallerIsSystem();
12   ArrayList<String> restrictions = new ArrayList<>();
13   for (SecureSpacesExtension extention : this.mExtensions.values()) {
14     for (UserRestriction restriction : extention.userRestrictions) {
15       if (restriction.deviceOwnerOnly) {
16         restrictions.add(restriction.name);
17         ...
18   return restrictions;
19 }
20 public class SecureSpacesExtension {
21   public ArrayList<UserRestriction> userRestrictions;
22 }
23 public class UserRestriction {
24   public String name;
25   public boolean deviceOwnerOnly;
```

Listing 6.2: Inconsistency in `SecureSpacesService`

Figure 6.3 illustrates the AC dependency graph corresponding to the target variable `mExtensions` in Listing 6.2. Next, we walk through this example to describe how Algorithm 1 creates the AC dependency graph:

- The category of the target variable `mExtensions` is determined to be a *COLLECTION* (line 4). Therefore, two synthetic child nodes, `ALL` and `SOME`, are created and connected to the parent `mExtensions` using a *collectionOf* edge (line 29 - line 30). (nodes omitted from the figure for brevity[4])
- The inner type of `mExtensions` is determined to be `<String,SecureSpacesExtension>` (line 24). The String key is determined to be a *SIMPLE* category. A corresponding node will be created (not shown for brevity).
- `SecureSpacesExtension` is determined to be a *COMPLEX* category. The algorithm resolves its member fields and creates corresponding nodes. One such node represents the member `userRestrictions` which is in turn determined to be of a *COLLECTION*

[4]Similar synthetic nodes are shown later for `userRestrictions`.

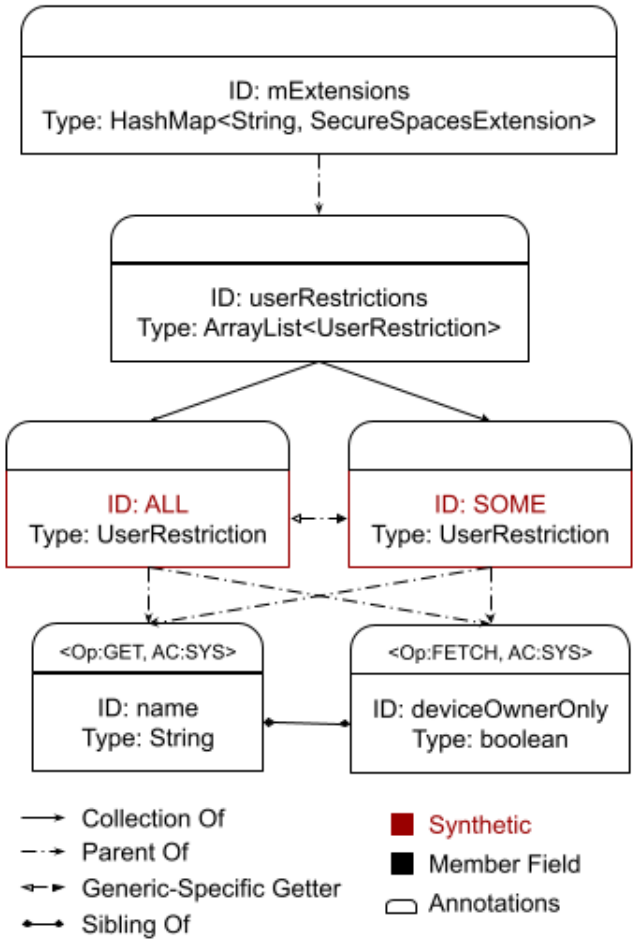

Figure 6.3: AC Dependency Graph for `mExtensions`

(line 4).

- The process similarly creates two synthetic members, `ALL` and `SOME` nodes of inner type `UserRestriction`, and connects them to the parent `userRestrictions` using *collectionOf* edge (line 29 - line 30).
- `UserRestriction` is determined to be of *COMPLEX* category (line 4). Two nodes, representing its member fields *name* and *deviceOwnerOnly*, are then created and connected to the containing class object node `UserRestriction` using *containsMember* edge (line 15). The construction finishes at this stage since there are no more nodes to be resolved.

Table 6.2: Operation and granularity semantics

	Operation/Granularity	Static Pattern
Operation	GET	Read and return field
	SET	Set field using parameters
	FETCH	Read field without returning it
	MODIFY	Modify field without using parameters
	ADD	Add member to collection
	REMOVE	Remove member from collection
	INDEX_EXISTS	Return existence of a collection index
	VALUE_EXISTS	Return existence of a collection value
Granu	ALL	Operating on all fields
	SOME	Operating on a subset of fields

6.4.3 Step 3: Annotation

In addition to storing the identifier and type information associated with each target variable, the nodes also store access control annotations required to operate on it. For example, a node might have the access control annotation: *[Read,* Normal*], [Write,* Dangerous*]*, indicating that reading the target variable requires a Normal permission, while writing to it requires a Dangerous permission. Ariadne extracts access control annotations of a given target variable by performing static analysis on Android APIs.

Target Operations and Granularity. Utility APIs operating on composite data types provide various operational semantics (e.g., check if an element is in a Collection, return an element in a Collection) which may carry access control implications (Section 6.2). Modeling the entirety of operation semantics is infeasible to develop (given the domain-specific nature of Android APIs) and even to analyze (unclear access control implications). Hence, we focus our analysis scope on a predefined set of operation semantics that can have *intuitive and direct* access control implications[5].

Table 6.2 lists the operation and granularity semantics that we have compiled manually and using our domain knowledge. They span ten categories, and can be detected by relying on the sample static patterns listed in column 2. Observe that some of the patterns require precise analysis.

Access Control Extraction. Ariadne collects access control-related information using

[5]These include both certain and uncertain implications.

path-insensitive analysis. It performs a forward control-flow analysis on the API's inter-procedural control flow graph (ICFG) and pinpoints the conditional branches on which *a target sink* is control dependent – where a target sink corresponds to an operation performed on a target variable. The analysis follows the traditional approach proposed by Kratos [111] and Axplorer [15] to resolve and identify conditional and non-conditional statements traditionally used for access control enforcement. Finally, the identified access control checks are normalized using the approach proposed by AceDroid [1]; where each access control check is categorized based on the level of permission it requires: (i) Normal, (ii) Dangerous, (iii) Signature, and (iv) System. access control checks that do not rely on permissions are categorized as System since they check for a privileged user.

Finally, the static analysis results are consolidated to form an access control annotation for a given node.

Initial propagation of access control annotations. Once a node is annotated, Ariadne propagates the annotations to nodes connected via deterministic *containsMember* edges. Specifically, Ariadne traverses backwards from the node's AC dependency graph to iteratively propagate the annotation towards the root. The traversal stops if a non-deterministic edge is encountered.

Ariadne resolves potential annotation conflicts between a class instance node c and its member field node m as follows:

- If c enforces access control equivalent or stronger than m, keep c's annotation; propagate it to connected nodes henceforth.
- Otherwise, overwrite c's annotation with that of m and continue propagating annotations towards root (fixing them en route). *Signal an access control inconsistency in this case.*

This ensures that access control required to operate on a target variable is equivalent or stronger than that of its fields or elements.

6.5 Non-deterministic Propagation via Flooding

To predict missing access control annotations, we treat the AC dependency graph as a *flow network* [97, 104], using controlled flooding to propagate annotations through non-deterministic edges.

6.5.1 Step 4: Assigning Edge *reduction_factor*

Table 6.3: Formal notations and their definitions

Term	**Notation and Definition**
Protection	$p \in \{\text{NONE}, \text{NORMAL}, \text{DANGEROUS}, \text{SYS\_OR\_SIG}\}$[6]
Operation	$op \in \{\text{GET}, \text{SET}, \text{ACCESS}, \text{MODIFY}, \text{ADD}, \text{REMOVE}, \text{IND\_EXISTS}, \text{VAL\_EXISTS}\}$
AC dependency graph nodes	GN
AC dependency graph edges	GE
AC(n, p, op)	Node $n \in GN$ requires `protection` p for `operation` op
relevance_weight (n, p, op)	*relevance_weight* of AC for node n

Table 6.3 presents the notations we use to define the non-deterministic edge labels (Table 6.5). `Protection` p denotes a normalized form of Android access control [7]. We use op to denote an `Operation`, which refers to the operation semantics (Table 6.2).

Table 6.5 shows the *reduction_factor* for each type of *non-deterministic* edge. Each row lists the condition(s) that identifies a *non-deterministic* edge.Table 6.4 lists the access control relations that correspond to *the non-deterministic* edges.

Logical relations. Edges propagate access control annotations by relying on Java object access control relations. Edges 13 and 14 state that access control annotations of a class instance node n_1 can be propagated to its member field node n_2 with *reduction_factor* rf. If n_1 is the only member of n_2 (13), we can intuitively link the access control implications with a high confidence, hence $rf = 0.95$. Otherwise, if n_2 has more than one member fields (14), we are less certain about the access control implication, hence we set $rf = 0.6$. Edge 15 identifies an edge between two sibling nodes, and sets $rf = 0.6$ given the uncertainty of the access control relation. Edge 16 states that nodes whose types extend the same superclass may share a similar protection. The edge is bidirectional with $rf = 0.7$.

[6]We borrow the ordering of AC from [1] as: NONE < NORMAL < DANGEROUS < SYS_OR_SIG

[7]We use the normalization scheme defined by AceDroid, which maps a permission enforcement (or other checks) to permission protection levels.

Table 6.4: Definitions of access control relations

Facts and Observation	Description
$contains(n_1, n_2)$	Node n_1 contains member field node n_2
$one\_member(n)$	Node n has one member field
$m\_members(n)$	Node n has more than one member field
$specific\_generic(n_1, n_2)$	Nodes n_1 and n_2 are `SOME` and `ALL` members of same collection, respectively
$sibling(n_1, n_2)$	Nodes n_1 and n_2 are defined by the same (super) class
$same\_supertype(n_1, n_2)$	Nodes $n1$ and $n2$ extends the same (super) type

Contextual relations. To compensate for the uncertainty of logical relations, Ariadne considers *naming hints* to form additional edges.

For example, two sibling nodes with similar variable names are more likely to have similar access control annotations. Similarly, a member field node is likely to inherit its containing class node's protection if their names are similar. We introduce the edge in Edge 17 to encode this observation. The *reduction_factor* is a function of $sim(n_1, n_2)$, a similarity score of n_1 and n_2. We calculate the similarity score using four features: (i) variable names, (ii) types, (iii) associated modifiers (e.g., public, private) and keywords (e.g., static, volatile), and (iv) names of methods operating on the field. (Section B.1)

Granularity semantics relations. These edges allow inferring access control based on the operation granularity. Particularly, the relations *Specific-to-generic* (for *Collection* types) imply that if a subset of the collection members requires an access control, then the collection itself requires a similar access control. The opposite implication is uncertain. In our graph representation, this relation corresponds to the edge connecting <`ALL`, `SOME`> (Subsection 6.4.2).

Ariadne identifies edges between nodes n_1 and n_2 of type `ALL` and `SOME` as depicted in Edge 18 and 19, where the access control implication is propagated with $rf = 0.95$ from n_1 to n_2 and with $rf = 0.4$ in the opposite direction.

Table 6.5: Edge definitions and their *reduction_factor* values

Edge ID	Edge Label	Condition	***reduction_factor*** (rf)
13	Class-1-Member	$n_1 \in GN \wedge n_2 \in GN \wedge contains(n_1, n_2) \wedge one\_member(n_1)$	0.95
14	Class-m-Members	$n_1 \in GN \wedge n_2 \in GN \wedge contains(n_1, n_2) \wedge m\_members(n_1)$	0.6
15	Siblings	$n_1 \in GN \wedge n_2 \in GN \wedge sibling(n_1, n_2)$	0.6
16	shareSuper Type	$n_1 \in GN \wedge n_2 \in GN \wedge same\_supertype(n_1, n_2) \wedge n_1 \neq n_2$	0.7
17	Similarity	$n_1 \in GN \wedge n_2 \in GN \wedge (contains(n_1, n_2) \vee sibling(n_1, n_2))$	$0.6 \times sim(n_1, n_2)$
18	Specific-Generic	$n_1 \in GN \wedge n_2 \in GN \wedge specific\_generic(n_1, n_2)$	0.95
19	Generic-Specific	$n_1 \in GN \wedge n_2 \in GN \wedge specific\_generic(n_2, n_1)$	0.4

6.5.2 Step 5: Propagating Access Control Annotations

Algorithm 2 details the flooding steps we follow to propagate access control annotations (1) across different nodes and (2) within the same node (across different operations).

Cross-nodes access control propagation assigns access control annotations of nodes (line 4). It adjusts corresponding *relevance_weight* based on the *reduction_factor* of the edges connecting them. line 4 combines *relevance_weight* of multiple access control annotation flows from incoming edges using a noisy OR operation [114], ensuring that the combined value is less than 1.

In-node access control propagation propagates access control annotations across different operations within the same node (Table 6.2). This allows to express access control implications between common operations. For example, an access control annotation for a *GET* operation implies that the *MODIFY* operation likely requires an annotation that is at least as strong.

line 8- line 11 adjusts *relevance_ weight* of the access control annotations of two operations op_1 and op_2 within a node. As shown, the steps use a function $op\_sem(op_1, op_2)$ to estimate the *reduction_factor* of the uncertain access control implication from op_1 to op_2. We estimate *op_sem* using our domain knowledge.

Output of the algorithm is a list of access control annotations per operation, ranked by *relevance_ weight*, for each node in the AC dependency graph. Ariadne then propagates annotations to corresponding APIs (that is, those operating on data holders denoted by the nodes). We use an empirically-chosen *cut-off* value (Subsection 6.6.6) to select highly relevant access control annotations (i.e., where *relevance_ weight* > *cut-off).*

Ariadne flags potential inconsistencies when an API enforces a weaker access control than Ariadne's reported access control annotations.

Algorithm 2: Access Control Propagation through Flooding

Input : *ACDepGraph* with node set = *GN*
Output : Set of final *relevance_ weight* values for annotations of data holders
Requires :

1. neighbours(n): Returns the list of nodes connected to n in *ACDepGraph*.
2. flood(n_1, n_2): $\forall AC \in$ annotations of n_1, *relevance_ weight*(n_2, p, op) = *relevance_ weight*$(n_2, p, op) \oplus$ *relevance_ weight*$(n_1, p, op) \times$ *reduction_factor*.
3. allOps(n): A list of all operations performed on node n.
4. $op\_sem(op_1, op_2)$: Returns *reduction_factor* corresponding to access control implication between operations op_1 and op_2.

```
1  while True do
2      for node ∈ GN do
3          for n ∈ neighbours(node) do
4              flood(node, n)
5          end
6      end
7      for n ∈ GN do
8          for ⟨op1, op2⟩|op1 ∈ allOps(n) ∧op2 ∈ allOps(n) ∧op1 ≠ op2 do
9              relevance_ weight (n, p, op2) = relevance_ weight (n, p, op2)⊕ relevance_ weight (n, p,
                 op1)× op_sem(op1, op2))
10             relevance_ weight (n, p, op1) = relevance_ weight (n, p, op1)⊕ relevance_ weight (n, p,
                 op2)× op_sem(op2, op1))
11         end
12     end
13     if relevance_weight did not change then
14         break
15     end
16 end
```

6.6 Implementation and Evaluation

We develop a prototype of Ariadne consisting a static analyzer, built using WALA [103], and Java based implementations of Algorithm 1 and Algorithm 2. The static analyzer is responsible for modeling data holders and underlying relations into AC dependency graphs. It processes custom Android frameworks, identifies data holders, and builds corresponding (partially-annotated) access control dependency graphs. The flooding algorithm then propagates access control annotations and outputs the likely access control inconsistencies. Our analysis is performed on a machine with an 8-core CPU (Intel(R) Core-i7-10510U @ 1.80GHz) and 16G main memory.

6.6.1 Test ROMs

To test our prototype, we analyze 13 ROMs, collected from public online repositories [42, 44] and physical devices. The samples range from Android versions 8 to 14 and correspond to two AOSP ROMs and 11 custom ROMs, from eight popular vendors; Samsung, Xiaomi, ZTE, Nubia, Oppo, Amazon, Tecno, and Vivo. Column 1 in Table 6.6 details the device model and version for each ROM.

Breakdown of vendor customization. Columns 2 and 3 report the total number of defined APIs and of vendor-customized ones, respectively. The number of custom APIs differs across vendors, ranging from as little as 361 (13.8%) in Xiaomi Mix 2S to 4218 (72.4%) in Samsung ZFlip. Column 7 reports the number of (AOSP and custom) variables reachable from the APIs, while column 8 reports their custom portion – i.e., vendor-defined variables. The latter are extensive, reaching up to 45027 (68.7%) in Samsung ZFlip underscoring the pervasiveness of the practice of data customization.

6.6.2 Impact of Target Selection Criteria

Column 9 shows the number of variables after applying the selection criteria Subsection 6.4.1, along with the percentage reduction (i.e., the number of reduced variables over the total number of recovered custom fields). As listed, the criteria lead to a substantial reduction of variables, averaging 94.5% for all analyzed ROMs.

False negatives A downside of the selection criteria is the possibility of false negatives. Given the absence of ground truth, we resorted to manual analysis to estimate FNs, carried out by two authors, who analyzed 5% of randomly-selected discarded variables (812 out of

Table 6.6: ROMs analysis statistics

(a) Part 1: API Statistics

	1. ROM	2. APIs	3. Custom APIs	4. Covered APIs	5. Inconsistent APIs	6. Correct Predictions
AOSP	Pixel 5 (v12)	1513	-	326 (21.5%)	10	315 (96.6%)
	Pixel 6 Pro (v13)	944	-	244 (25.8%)	5	238 (97.5%)
Custom	Xiaomi Mix 2S (v8)	2603	361	109 (30.1%)	3	-
	Amazon Fire HD (v10)	2896	821	228 (27.7%)	8	-
	ZTE Axon 40 (v12)	1995	514	115 (22.3%)	12	-
	Nubia Z50 Ultra (v13)	2076	482	107 (22.1%)	12	-
	Samsung A3058 (v10)	5888	2881	1361 (47.2%)	45	-
	Samsung Z Flip (v13)	5822	4218	1691 (40%)	35	-
	Samsung Tab S9+ (v14)	3767	2177	1203 (55.2%)	19	-
	Oppo A16S (v12)	1929	423	50 (11.8%)	2	-
	Oppo OP574FL1 (v14)	2010	512	86 (16.7%)	3	-
	Vivo V2309 (v14)	3507	849	167 (19.6%)	6	-
	Tecno Spark (v13)	3596	2002	720 (35.9%)	18	-

(b) Part 2: Field and Edge Statistics

	1. ROM	7. Fields	8. Custom Fields	9. Analyzed Fields (% Reduction)	10. Edges Generated
AOSP	Pixel 5 (v12)	17656	-	993 (94.3%)	1438
	Pixel 6 Pro (v13)	16935	-	1085 (93.5%)	1745
Custom	Xiaomi Mix 2S (v8)	27726	2747 (9.9%)	1736 (93.7%)	4041
	Amazon Fire HD (v10)	44603	16681 (37.4%)	2720 (93.9%)	4065
	ZTE Axon 40 (v12)	23988	6044 (26.1%)	1279 (94.6%)	2489
	Nubia Z50 Ultra (v13)	24410	7456 (30.5%)	1361 (94.4%)	2786
	Samsung A3058 (v10)	65160	26209 (40.2%)	3238 (95%)	10170
	Samsung Z Flip (v13)	65483	45027 (68.7%)	3083 (95.2%)	7776
	Samsung Tab S9+ (v14)	33934	14523 (42.7%)	1597 (95.2%)	11478
	Oppo A16S (v12)	22760	5538 (24.3%)	1179 (94.8%)	2085
	Oppo OP574FL1 (v14)	25414	6634 (26.1%)	1241 (95.1%)	5075
	Vivo V2309 (v14)	47485	13819 (29.2%)	2897 (93.8%)	6021
	Tecno Spark (v13)	42985	19172 (44.6%)	2290 (94.6%)	16681

17295 variables for AOSP and 1812 out of 38254 variables for custom ROMs) by checking whether they are connected to data holders via relations carrying access control implications (Table 6.1). We did not find any such cases. To estimate the maximum possible FN rate, we use Clopper-Pearson Exact Method [22] for binomial proportions with zero observed FNs. We calculate the confidence upper bounds using the formula $1 - \alpha^{1/n}$, where n is number of variables (2624 in our case), and α (0.05 for a 95% confidence interval, and 0.01 for 99%) is the probability of a FN rate exceeding the computed upper-bound, leading to 0.1107% and 0.175% confidence upper bounds on the FN rate.

Therefore, we can conclude that the selection criteria are unlikely to cause FNs.

6.6.3 Inconsistency Detection Accuracy

Accuracy estimation. Due to the lack of ground truth for custom ROMs, we estimate the accuracy of Ariadne using AOSP (in line with prior work[135, 128, 35]). Specifically, we predict a normalized protection for each *target API* – i.e., those operating on data holders (Subsection 6.4.1), and compare it against its actual access control enforcement, implemented by the API. If the prediction is equal to the latter, we flag the case as correct.

Our experimental setup is as follows: for each target API, we run Ariadne on all APIs except the target API to (a) extract access control enforced to protect reachable data holders, (b) extract access control relations that connect these data holders to other data holders, and (c) generate the access control dependency graph. Ariadne then performs controlled flooding to compute normalized access control protections with *relevance_ weight* for the data holder(s) reachable from the target API. The access control protections are recommended per operation. Hence, we select the protection(s) corresponding to the target API's operation on the data holders.

Results. Column 4 in Table 6.6 reports the number of APIs that contain a data-holder-related ('data-related' for brevity) operation. These are the APIs covered by Ariadne. For custom ROMs, this number is for custom APIs. Overall, the ratio of data-related APIs differs across vendors, ranging from 11.8% in Oppo A16S and reaching up to 47.2% in Samsung A2058, averaging 27.8% of the custom, and 23.6% of AOSP APIs.

Column 6, Rows 1-2 report the number of AOSP APIs with a correct access control prediction (this metric is not reported for other ROMs due to the absence of ground truth). The accuracy ranges from 96.6% in AOSP Pixel 5 (v12) to 97.5% in AOSP Pixel 6 Pro (v13).

AOSP False positives. We consider the cases where Ariadne projects an access control recommendation that is stronger than the implementation as false positives. As shown,

our tool has 2.95% FP rate. We manually investigate the reported FPs and identify the following root causes:

Over-approximation of access control initial beliefs. To extract the initial beliefs, Ariadne relies on the approximation that an access control protects all data holder operations that *are control-dependent* on the access control check. This approximation may lead to inaccuracies as not all control-dependent operations are necessarily related to the access control (consider the case where an AC SYS_OR_SIG (Table 6.3) precedes a sensitive native method and an insensitive SET (Table 6.2) operation of var var_1, Ariadne will construct an inaccurate initial belief: [var_1,SET] requires AC SYS_OR_SIG). Poirot [35] can address this challenge by reasoning about protection targets. Hence, it is beyond the scope of this work. We discuss in Section 6.7 how an approach that combines both solutions (Poirot and Ariadne) can address this limitation effectively.

Inaccurate runtime type resolution. Resolving Java runtime types statically is a hard problem [105, 81, 7]. We use WALA to create context-sensitive callgraphs using 0-1CFA algorithm; however, runtime types of some variables cannot be determined accurately, which may affect the accuracy of the AC dependency graph construction.

6.6.4 Inconsistency Detection in Custom ROMs

We found that 17% of framework dex files (containing system service implementations) in the 11 custom ROMs cannot be correctly decompiled or processed by WALA (in particular, the Vivo ROM). Among the rest, Ariadne discovered a total of 163 data-driven inconsistencies, out of which 90 are unique. Column 5 in Table 6.6 shows a detailed breakdown. Every single ROM contains data-driven inconsistencies, ranging from two in Oppo A16S to 45 in Samsung ZFlip.

Due to the lack of ground truth in custom ROMs, we resort to manual analysis to evaluate the reported positives. Two authors inspected the decompiled code of each reported case and labeled it into one of three categories: (1) *true positive* – the API indeed lacks an access control enforcement along the path to a data holder and thus can be exploited, (2) *false positive* – the data holder accessed by the API is not sensitive and thus does not require AC, and (3) *unresolved positive* – it is unclear whether the API should enforce AC because of a proprietary feature, necessitating further examination by developers familiar with the codebase (i.e., vendor framework developers). Figure 6.4 shows a breakdown of positives: On average, 11.8% are FPs (with similar root causes as AOSP), 32.8% unresolved positives, and 55.1% TPs.

To demonstrate the security impact of TPs, we built end-to-end proof-of-concept exploits, listed in Table 6.7 with a range of security/privacy consequences: disabling highly-critical system packages such as the “android" package itself in Tecno, compromising a critical security policy in Samsung, launching random app Activities with system privilege in Vivo (i.e., an app can trigger protected Activities without requiring a permission), inferring currently running apps on the phone, and others. We have responsibly reported these to the vendors. At the time of writing, 11 are acknowledged, and two are under verification. We manually analyzed each and confirmed that seven are exclusively detected by Ariadne due to their data-driven nature. We describe one case in detail in Section 6.9.

6.6.5 Availability and Impact of Edge Types

An essential requirement for the success of Ariadne is the availability of (deterministic and non-deterministic) edges in the AC dependency graph. Column 10 in Table 6.6 reports the total number of detected edges, which ranges from 1438 in AOSP up to 16681 in Tecno Spark. The custom ROMs (rows 3-13) introduce more services (on average 719 per ROM vs 378 in AOSP), APIs (up to 4218 in Samsung Z-FLIP), and data-holders (on average 2056 per ROM vs 1039 in AOSP), contributing to significantly more edges. We further study the relative impact of each edge type on access control predictions by examining its overall contribution to the total number of generated edges. The distribution follows a similar pattern across ROMs (Section B.3) – where the similarity edges are the most common, followed by Class-1-Member. Other edges are less prevalent; nonetheless, helpful in detecting true inconsistencies (Section 6.9).

Table 6.7: Summary of Discovered Vulnerabilities

ROM	Service API	Security Implication	Report Status
Tecno Spark	PackageManagerService .enabledPackage	Enable / disable arbitrary (critical) packages – Allows device breakdown	Acknowledged
Vivo V2309	VivoWmsImpl .startDoubleWpsSplitScrean	Launch arbitrary protected app Activities with system privilege – Allows bypassing app component protection	Acknowledged
Vivo V2309	ActivityTaskManagerService .getLastResumedActivityPkgName	Identify recently running packages	Reported
Samsung A3058	PersonaPolicyManagerService .setRCPDataPolicyForFota	Manipulate and compromise the integrity of Samsung's data policy	Acknowledged
Xiaomi MIX2S	SecureSpacesService .getUserRestriction	Expose privileged policy data 'user restrictions' to unauthorized users	Acknowledged*
Xiaomi MIX2S	SecureSpacesService .getExtensions	Expose privileged policy data 'Secure space extension name' to unauthorized users	Acknowledged*
Samsung TabS9+	CustomFrequencyManagerService .restrictApp	Restrict arbitrary apps	Acknowledged+
Samsung TabS9+	RemoteAppModeService .setLTWProtocolVersion	Set LTW protocol version	Acknowledged+
Tecno Spark	PowerManagerService .triggerBatterySaver	Allows trigerring battery saver mode	Reported
Samsung ZFlip	KnoxCustomManagerService .getAutoCallNumberList	Get list of auto call numbers setup by a user	Acknowledged
Amazon Fire HD	AmazonAccessibilityManagerService .magnificationCanvasAddLine	Manipulate the magnification screen for accessibility services	Acknowledged
Amazon Fire HD	AmazonAccessibilityManagerService .magnificationCanvasAddRect	Manipulate the magnification screen for accessibility services	Acknowledged
Amazon Fire HD	AmazonAccessibilityManagerService .magnificationCanvasClear	Remove the magnification screen for accessibility services	Acknowledged

*The API is removed in the latest versions.

+ The vulnerability was flagged as duplicate because it was already reported by a different party.

6.6.6 Impact of Cut-off *relevance_ weight*

Ariadne relies on a cut-off threshold to select high-confidence access control recommendations (Subsection 6.5.2). We evaluate the impact of this threshold on the overall results. We

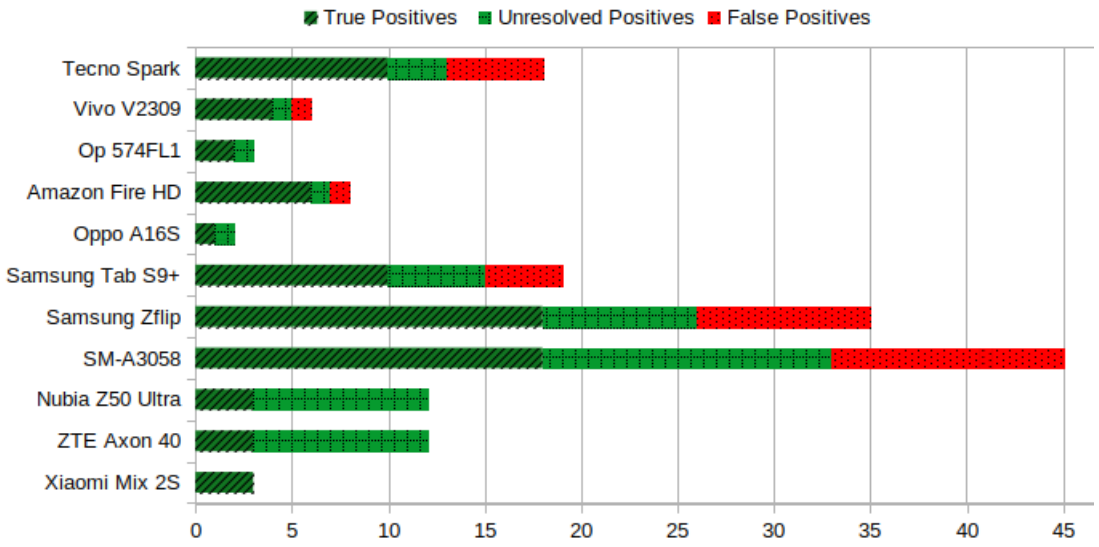

Figure 6.4: Categorization of positives reported by Ariadne

Table 6.8: Positive Reports for Samsung ZFlip

Cut-Off	True	False	Unresolved	Total
0	19 (45.2%)	14 (33.3%)	9 (21.4%)	42
0.25	18 (51.4%)	9 (25.7%)	8 (22.8%)	35
0.4	16 (50%)	9 (28.1%)	7 (21.8%)	32

select three cut-off values: 0, 0.25 and 0.4, and calculate the total number of true positives, unresolved positives and false positives for Samsung ZFlip (Table 6.8). 0.25 represents the optimum value, providing a balance.We repeated this experiment for the rest ROMs, which revealed a similar trend. We omit the details for brevity.

6.6.7 Impact of Variations in *reduction_factor*

We selected *reduction_factor* values using domain knowledge (Subsection 6.5.1) to reflect our degree of confidence in each access control implication. A higher *reduction_factor* indicates stronger confidence (i.e., lower uncertainty), and vice-versa. To evaluate the sensitivity of Ariadne to variations in *reduction_factor*, we changed its value in four experiments:

Exp 1. Rule 13 from 0.95 to 0.9

Table 6.9: Impact of *reduction_factor* on inconsistencies

ROM	Exp 1	Exp 2	Exp 3	Exp 4
Pixel 5 (v12)	10	10	10	10
Samsung ZFlip (v13)	35	34	35	34

Exp 2. Rule 13 from 0.95 to 0.8

Exp 3. Rule 17 from 0.6 to 0.65

Exp 4. Rule 19 from 0.4 to 0.2

We counted the number of inconsistencies reported for two sample ROMs under the modified values (Table 6.9). The results show that changes in *reduction_factor* have a negligible impact on the reported inconsistencies. Only one (1/35) of the inconsistencies was missed in Samsung ZFlip. Our manual inspection showed that the affected case had limited data-driven access control implications, suggesting that the *reduction_factor* has a more pronounced effect when the number of edges is low.

6.7 Comparison with Prior Approaches

We compare Ariadne with inconsistency detection approaches based on: i) convergence and (ii) probabilistic-inference.

6.7.1 Convergence-based Approaches

Convergence-based approaches detect inconsistencies by comparing access control enforced along different paths accessing the same resource. Kratos [111], ACMiner [64] and AceDroid[1] look for inconsistencies within the framework, by identifying APIs converging on a shared resource.

These approaches are limited in their ability to detect data-driven inconsistencies, since they can only detect cases where two APIs converge on a common sink (e.g., Java or native method invocation, field update, throw instructions). When dealing with data-driven inconsistencies, they are restricted to detecting only those where the convergence point is a field get or set operation. E.g., they cannot reason about the consistency of access control in cases where (i) two APIs operate on the same variable differently (e.g., one reads a list, while another checks if an item exists in the same list) and (ii) two APIs access

highly-related fields (e.g., one reads a member field while another reads its containing class instance).

To demonstrate this, we compare Ariadne against ACMiner [64], a state-of-the-art convergence-based inconsistency detection. We select ACMiner since its detailed inconsistency results are publicly available for a sample Android ROM (Nexus 5X – v7.1.1)[8]. We run Ariadne on the same ROM and compare our results to ACMiner's. We investigate Ariadne's FPs through manual investigation. ACMiner reports 28 (manually-confirmed) access control inconsistencies while Ariadne reports 26 inconsistencies, which partially overlap.

Missing Inconsistencies. Five are exclusive to ACMiner. Ariadne missed them because they are not data-driven, i.e., the vulnerable APIs do not operate on a data holder.

New Inconsistencies. Ariadne catches eight exclusively. We confirmed manually that ACMiner cannot detect them because the responsible APIs did not converge on a handled sink (e.g., method invocation, direct field update). Rather, the APIs accessed data holders that were connected to other protected data holders via implicit access control relations, like sharing the same super type, similarity, and operation relations. Ariadne caught those due to its ability to model such relations, therefore, these constitute the problem space exclusively solved by Ariadne. E.g., Ariadne identified that `UserManagerService.getUserRestrictions` (v7.1.1) requires a SYSTEM access control; which was added in versions starting from v8.0.

Rediscoveries. 18 are detected by both tools. This is possible when *the convergence point* of two inconsistently-protected APIs is a *direct* data holder access using the same operation. A common example is `getInstalledApplications` from `PackageManagerService`.

6.7.2 Probabilistic Approaches

Two probabilistic-inference approaches, Poirot [35] and Bluebird [128], have been proposed to alleviate some shortcomings of pure convergence-based tools. They both account for the uncertainty surrounding access control specification inference. Poirot suppresses FPs of convergence-based technique by pinpointing accurate targets of access control enforcement. It relies on *hints* such as control-dependencies, naming, and trigger-conditions. Bluebird [128] on the other hand, detects access control gaps in framework APIs by learning from system apps. It is limited in scope as it cannot tackle APIs not invoked by apps.

None of the prior tools models implicit access control relationships among Java objects or those arising from the type / granularity of underlying operations. Ariadne thus complements

[8]ACMiner does not run on our collected ROMs.

them by capturing a new class of access control inconsistencies.

Ariadne draws inspiration from Poirot and Bluebird in modeling uncertainty. While prior works rely on probabilistic inference, Ariadne employs customized flooding to propagate uncertainty. Since Ariadne uses AC dependency graph, leveraging graph-based techniques enables its seamless integration into Ariadne's analysis pipeline. Unlike probabilistic inference — which relies on expensive computations of posterior marginals — flooding offers a scalable alternative, allowing Ariadne to efficiently tackle the sheer number of data-driven access control implication, by providing a tunable trade-off between accuracy and performance through iteration depth.

We compare Ariadne to Poirot and Bluebird to demonstrate the unique benefit of each approach. We run Ariadne on the same ROMs analyzed by Poirot and/or Bluebird – Amazon Fire HD(v10) and Samsung A3058.

Rediscoveries. Ariadne rediscovers 4 previously reported cases by Poirot and/or Bluebird. E.g., `setScheduleType` (originally from Bluebird), and `setAmazonFlags` and `removeAmazonFlags` (originally from Poirot).

New Inconsistencies. Ariadne discovers 22 new inconsistencies (i.e., not reported by either tool). We investigated them and concluded that they cannot be caught by Bluebird or Poirot since the responsible APIs were neither invoked by system apps, nor contained code constructs that Poirot relies on. Rather, the APIs accessed data holders connected to other protected data holders via implicit AC relations and operation semantics, therefore, these constitute the problem space exclusively solved by Ariadne.

Missing Inconsistencies. Ariadne misses 12 inconsistencies that Poirot and Bluebird detect. We manually analyzed the cases to confirm that they were not data-driven. One example is `AmazonAspService.command`, which includes no data-related operations. It only invokes a privileged native method.

6.8 Threats to Validity

Ariadne relies on logical and contextual relations derived from the AC semantics of 1) Java objects and 2) operations performed by Android APIs. The Android framework also includes APIs that do not target operations on the Java objects (e.g., APIs that solely behave as wrappers to native methods). Therefore such APIs do not provide any AC implications that can be leveraged by Ariadne. This results in Ariadne not being able to infer AC recommendations for such APIs.

As discussed in Subsection 6.6.3, Ariadne considers all field operations performed by an API as potential targets of AC. This is an over-approximation, as certain fields may not contribute significantly to the functionality of the API (e.g., lock variables used for synchronization). We consider identifying the actual target field of the API's enforced AC out of scope for Ariadne. Prior works such as Poirot [35] can be utilized to this end, making them complementary to our approach.

Finally, Ariadne relies on accurate points-to analysis to statically resolve runtime Java types of objects referenced by framework variables. This is a known hard problem [105, 81, 7]. Ariadne partially mitigates this limitation by using 0-1-CFA algorithm [103] to build context-sensitive callgraphs, as 0-1-CFA provides a balance between accuracy and speed. Other computationally expensive methods of creating more precise context-sensitive callgraphs (e.g., n-CFA) can be used to further mitigate this limitation, however, we make a design decision to rely on 0-1-CFA to allow scaling of Ariadne to complex frameworks.

6.9 Case Studies

Disabling (critical) packages. Tecno Spark defines an unprotected custom API `enabledPackage` in `PackageManagerService`, which allows enabling / disabling any package. The API modifies a member field `mPackages` of `mSettings`, a complex AOSP data holder. Ariadne infers that manipulating `mPackages` should be protected, since other similarly named sibling fields (in AOSP) require access control. To demonstrate the impact of the unprotected API, we invoke the API to disable "android" package, which corresponds to the OS itself. The PoC resulted in a complete / unrecoverable device breakdown. We reported the case to Tecno, who acknowledged it.

Need for object-sensitivity. Ariadne's object-sensitive analysis is crucial for reducing FPs. We demonstrate a case study where an object insensitive analysis would lead to a false inconsistency. ZTE introduces `setAlarmAlignType` and `setLockscreenCleanType` in `ZTEPowerTrackerService` (Listing 6.3). The former enforces an access control, whereas the latter lacks any. They access distinct data holders — `mZteAlarmController` (Line 1) and `mLockscreenCleanController` (Line 2) — both extending `BaseOptimizerController`. They invoke `setPkgValue` (Line 3) modifying `mSettingsMap` in the superclass (Line 2). An object-insensitive analysis will incorrectly identify convergence, and an inconsistency. Ariadne's object sensitivity correctly identifies the absence of convergence, and eliminates the FP.

```
1 private ZteAlarmController mZteAlarmController;
```

```
2 private ZteLockscreenCleanController mLockscreenCleanController;
3 public void setAlarmAlignType(String name, int type) {
4   enforceCallingOrSelfPermission("android.permission.DEVICE_POWER");
5  mZteAlarmController.setAlarmAlignTypeInternal(name,type);
6 }
7 public void setLockscreenCleanType(String name, int type) {
8   mLockscreenCleanController.setLockscreenCleanInternal(name,type);
9 }
```

Listing 6.3: ZTEPowerTrackerService with two APIs

```
1 class BaseOptimizerController {
2   private HashMap<String, Integer> mSettingMap;
3   public void setPkgValue(String name, int type) {
4     this.mSettingMap.put(name, Integer.valueOf(type));
5 }
6 class ZteLockscreenCleanController extends BaseOptimizerController {
7   void setLockscreenCleanInternal(String pkgName, int type) {
8     setPkgValue(pkgName, type);
9 }
10 class ZteAlarmController extends BaseOptimizerController {
11   void setAlarmAlignTypeInternal(String pkgName, int type) {
12     setPkgValue(pkgName, type);
13 }
```

Listing 6.4: Data holders used by ZTEPowerTrackerService

Chapter 7

Large-Scale Access Control Audit of Replicated Android APIs

This chapter is adapted from the paper [143] that presents a large-scale measurement study of code duplication in the Android framework, facilitated by RepFinder, a tool that infers the core functionality of an API and detects syntactically and semantically similar APIs using static program paths. RepFinder investigates a new source of customization vulnerabilities: *cloned framework APIs*, or *Replicas*, introduced through OEM copy-paste-edit practices. RepFinder develops a static-analysis pipeline tailored to identify Replicas, assess their access control enforcement, and study their security implications across a wide range of vendor ROMs.

7.1 Introduction

The practice of copy-paste-edit 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 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 [43]. Other research

suggests that significant vulnerabilities are introduced in apps through cloned code (from app and library-specific code) [144, 149, 146, 14, 136].

Despite the numerous efforts [24, 4, 39, 155, 153] 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 paper, 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 [45, 72, 76, 83, 131, 106], structural-based [130, 151, 156, 70, 137, 145], and deep-learning based techniques [70, 133, 137, 145]. In the context of Android apps, structural similarity (Program Dependence Graph (PDG)- based [24, 43] and Control Flow Graph (CFG)-based [144]) is the de facto 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 solution 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 significant amount of non-core logic but diverging in their core functionality 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,

renamed clones, near-miss clones, and semantic clones) [17]. 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 slow 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 wild 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[1]) 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 anonymous repository https://zenodo.org/records/11521246.

7.2 Background and Motivation

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

[1]masking the vendor's name until the vulnerability is patched

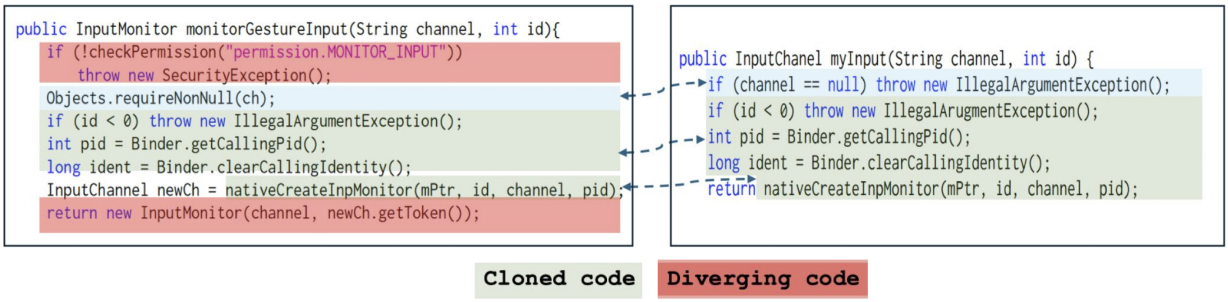

Figure 7.1: AOSP's `monitorGestureInput` and its vulnerable ZTE Replica `myInput`

7.2.1 Cloning Private APIs

Android OEMs 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 [111].

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 leads to Residuals [34]: 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.

7.2.2 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, 4, 39, 155, 153]. At the framework layer, we argue that cloning APIs could similarly introduce serious issues. For example, as discussed later (Subsection 7.3.1), 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 EOMs).

Motivating Vulnerable Replica: Figure 7.1 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.

7.3 How to Detect Replicas?

We observe that Replicas span the four traditional categories of clone types [17] with specific peculiarities.

7.3.1 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

```
public boolean setPowerMode(int mode, boolean on) {
    if (!mSystemReady) return false;
    enforceCallingOPermission("permission.DEVICE_POWER");
    setPowerModeInternal(mode, on);
}
```

```
public boolean setPowerModeChecked(int mode, boolean on) {
    if (!mSystemReady) return false;
    enforceCallingOPermission("permission.DEVICE_POWER");
    return setPowerModeInternal(mode, on);
}
```

(A) setPowerMode(..) and AOSP Replica setPowerModeChecked(..)

```
public long getUidStats(int uid, int type) {
    int callingUid = Binder.getCallingUid();
    if (callingUid != 1000 && callingUid != uid)
        return -1;
    return nativeGetUidStat(uid, type, checkBpfStatsEnable());
}
```

```
public long getUidStatsWithoutCheckUid(int uid, int type) {
    return nativeGetUidStat(uid, type, checkBpfStatsEnable());
}
```

Cloned code

(B) getUidStats(..) and Oppo Replica getUidStatsWithoutCheckUid(..) **Diverging code**

```
public void setBindAppWidgetPermission(String pkg, int grId, bool g) {
    enforceModifyAppWidgetBindPerm(pkg);
    ensureGroupStateLoadedLocked(grId);
    int pkgUid = getUidForPackage(pkg, grId);
    if (pkgUid < 0) return;
    Pair<Integer, String> pkgId = Pair.create(grId, pkg);
    if (g) mPkgsWithPerm.add(pkgId);
    else mPkgsWithPerm.remove(pkgId);
    saveGroupStateAsync(grId);
```

```
public boolean hasBindAppWidgetPermission(String pkg, int grId) {
    enforceModifyAppWidgetBindPermissions(pkg);
    ensureGroupStateLoadedLocked(grantId);
    int pkgUid = getUidForPackage(pkg, grantId);
    if (pkgUid < 0) return false;
    Pair<Integer, String> pkgId = Pair.create(grantId, pkg);
    return mPkgsWithPerm.contains(pkgId);
```

(C) setBindAppWidgetPermission(..) and Non-Replica hasBindAppWidgetPermission(..)

Figure 7.2: Cloning in API implementations

Figure 7.2(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 7.2(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 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 suboptimal 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[106], 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.*

This characteristic of Replicas narrows 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 7.2(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[2]. 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.

7.4 Our Approach: RepFinder

To detect and measure the prevalance of Replicas at large scale, we implement RepFinder, an analysis pipeline that combines static program analysis and NLP analysis. Figure 7.3 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.

[2]A semantic block denotes a data independent subgraph, as defined in [24]

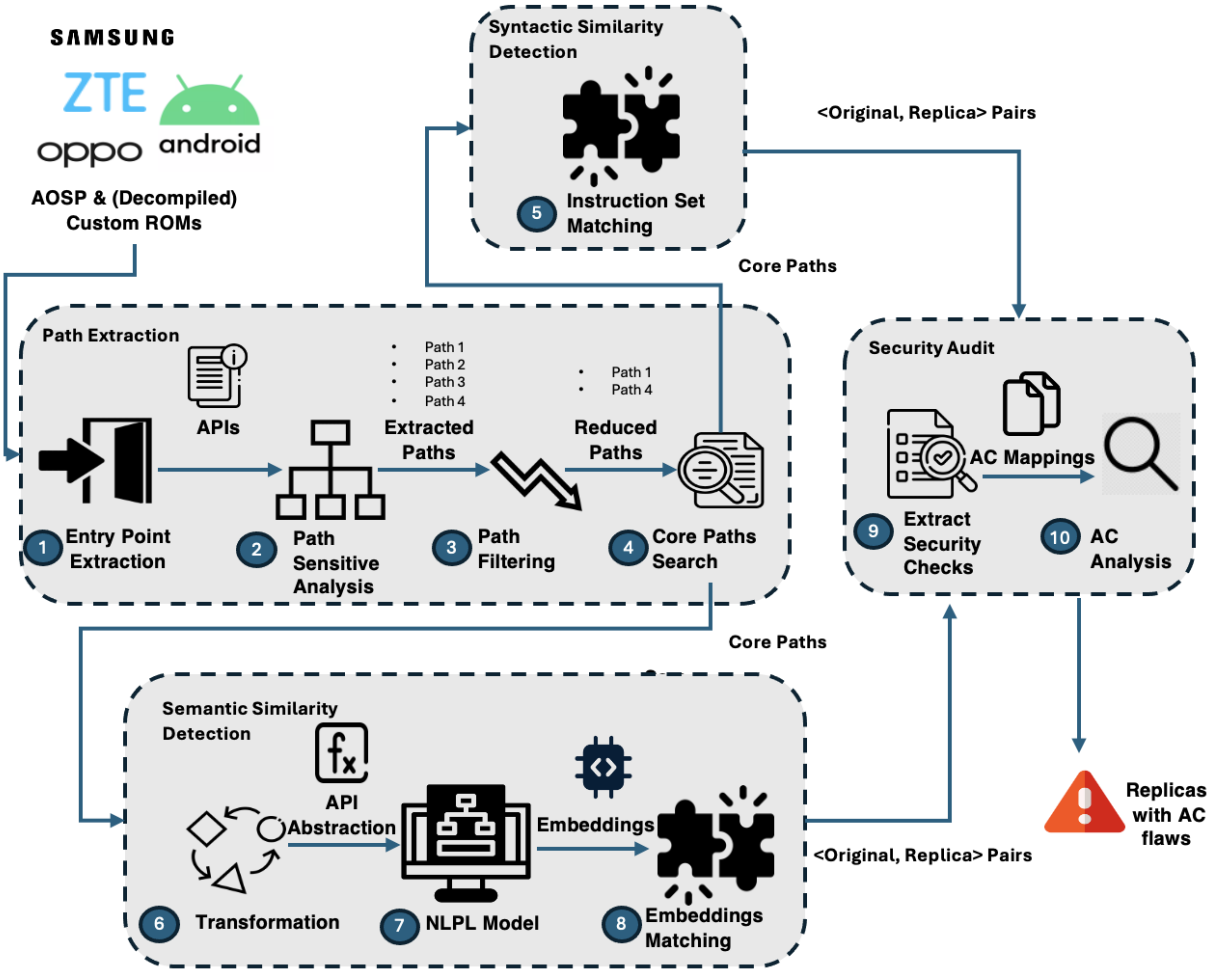

Figure 7.3: Workflow of RepFinder

7.4.1 Paths Extraction

Table 7.1: Paths identified in `removeData()` in `OplusDevicePolicy` – Oppo v.13

ID	Instructions	Class
1	`!isOplusReady()` → `return false`	Auxiliary
2	`isOplusReady()` → `!checkPermission()` → `return false`	Failed AC
3	`isOplusReady()` → `checkPermission()` → `mCache.get(mUserId) == null` → `Log.d("..")` → `return false`	Auxiliary
4	`isOplusReady()` → `checkPermission()` → `!(mCache.get(mUserId) == null)` → `ARG_2 == 1` → `mCache.get(mUserId).mOplusCustomizeData.remove(ARG_-1)` → `updateCustomizeFileHandle()` → `return true`	Core
5	`isOplusReady()` → `checkPermission()` → `!(mCache.get(mUserId) == null)` → `!(ARG_2 == 1)` → `mCache.get(mUserId).mOplusData.remove(ARG_1)` → `updateFileHandle()` → `return true`	Core

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 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 7.1 depicts a code snippet extracted from Oppo API `OplusDevicePolicy.removeData(..)`. For simplicity, we only lists the program paths identified by RepFinder at depth 1[3]. The paths are summarized in Table 7.1.

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; i.e., 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.

[3]the number of paths extracted at depth 3 is 7

```
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);
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 7.1: Oppo API `OplusDevicePolicy.removeData(..)`

7.4.2 Paths Filtering

Not all program paths in an API correspond to core logic functionality. For example, path 2 in Table 7.1 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 work [117, 92] 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, on 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 path 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.

7.4.3 Core Paths Selection

By this stage, RepFinder has reduced the set of candidate program paths. Observe the paths still include *auxiliary* paths which cannot be caught by the aforementioned rules (e.g., paths 1 and 3 in Table 7.1).

Inspired by prior work [128] 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, Replicas $>$ 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 θ, we consider P to be a core functional path of A if $Sim(P, M) \geq \theta$. We define Sim as follows: $Sim(P, M) = max(f(i_1, M), ..., f(i_n, M))$, 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[4]. To calculate the similarity, we generate and use various information for common instructions. For example, we generate the following instructions (we only show two 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 [66], 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 $emdedding(i_i)$.

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 paths to detect clones.

7.5 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.

[4]Dice similarity function can be used as well.

7.5.1 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 Subsection 7.8.1 for more details on the threshold selection.

Formally, given two APIs A_o and A_r, a similarity function Sim and a threshold θ, 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 $Sim(A_o.P,\ A_r.P) \geq \theta$.

7.5.2 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.

While Jaccard similarity is a simple and intuitive function for our task, it may to 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 (Subsection 7.4.3) alleviates the issue through reducing shared and noisy code. We further mitigate the issue by leveraging the notion of *core instruction*, defined in Subsection 7.4.3. 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 $|P_o \triangle P_r|$ contains at least one instruction i with $Dice_{Sim}(i, O) \geq \beta$, or $Dice_{Sim}(i, R) \geq \beta$, 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.

7.5.3 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 [66], OpenAI's codex embedding [23]), we use UniXcoder as it has achieved state-of-the-art performance on clone detection [66], and in other related tasks (e.g., code summarization [129] code to NL translation [27], code generation [32]).

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 path is in the form of an Intermediate Representation (IR) listing the instructions in the path. Such form is not amenable to UniXcoder, which is pretrained on code and NL. Therefore, we transform the IR statements to a near-code representation using basic transformation rules. Details are omitted for brevity.

7.6 Security Audit of Replicas

Table 7.2: ROM Dataset

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

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

7.6.1 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 (Subsection 7.4.1). 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.

7.6.2 Updates Propagation Consistency

Replicas add a layer of complexity to the already-complicated Android updates procedure [150]. To keep Replicas up-to-date, OEM developers need to carefully track APIs from which the Replicas were cloned and promply 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 Subsection 7.8.3.

7.7 Large Scale Measurement Study

We conduct a 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[103]. 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.

7.7.1 Dataset Collection

Our dataset spans 7 AOSP ROMs and 342 customized ROMs. We downloaded AOSP ROMs, which are necessary to filter public APIs [46], from the Android official repository [47]. We collected customized ROMs from FirmwareDrive[33], Sammobile[108], Samfrew [107], and AndroidDumps repository[44]. We wrote crawlers to automatically download the ROMs, when possible.

Dataset Characterization. Table 7.2 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 [115], our custom ROM dataset is representative of current Android OEMs. It covers the big players (Samsung, Xiaomi, Oppo, and Vivo), and includes others with smaller market shares (e.g., Motorolo, 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.

ROM Preprocessing. We used a suite of tools. We used a modified version of `SALT` tool [116] (handles kdz files), `simg2img`[16] (unpacks the super partition), `lpunpack`, `lpmake`[8], and `atcmd`[120] (combine and extract the system partitions). To mount the system partition, we use `imgtool`[80] and extract framework classes using `vdexextractor`[6], `baksmali`[74], `smali`[74], and `apktool`[123].

7.7.2 Analysis Landscape and Complexity

Customization Extent. RepFinder identified all together 44,794 private APIs. Table 7.2, 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

Table 7.3: Traces and Embeddings For ROMs

Vendor	Avg Paths per 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

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 7.3 reports the summary statistics for the path extraction phase (Subsection 7.4.1). 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).

Columns #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 7.3 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 7.3 is less than the path count (Column #5) because the semantic similarity detection (Subsection 7.5.3) is only concerned with *core functional paths*, which amount to two paths on average, per API.

7.8 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?

7.8.1 RQ1. Replicas Prevalence

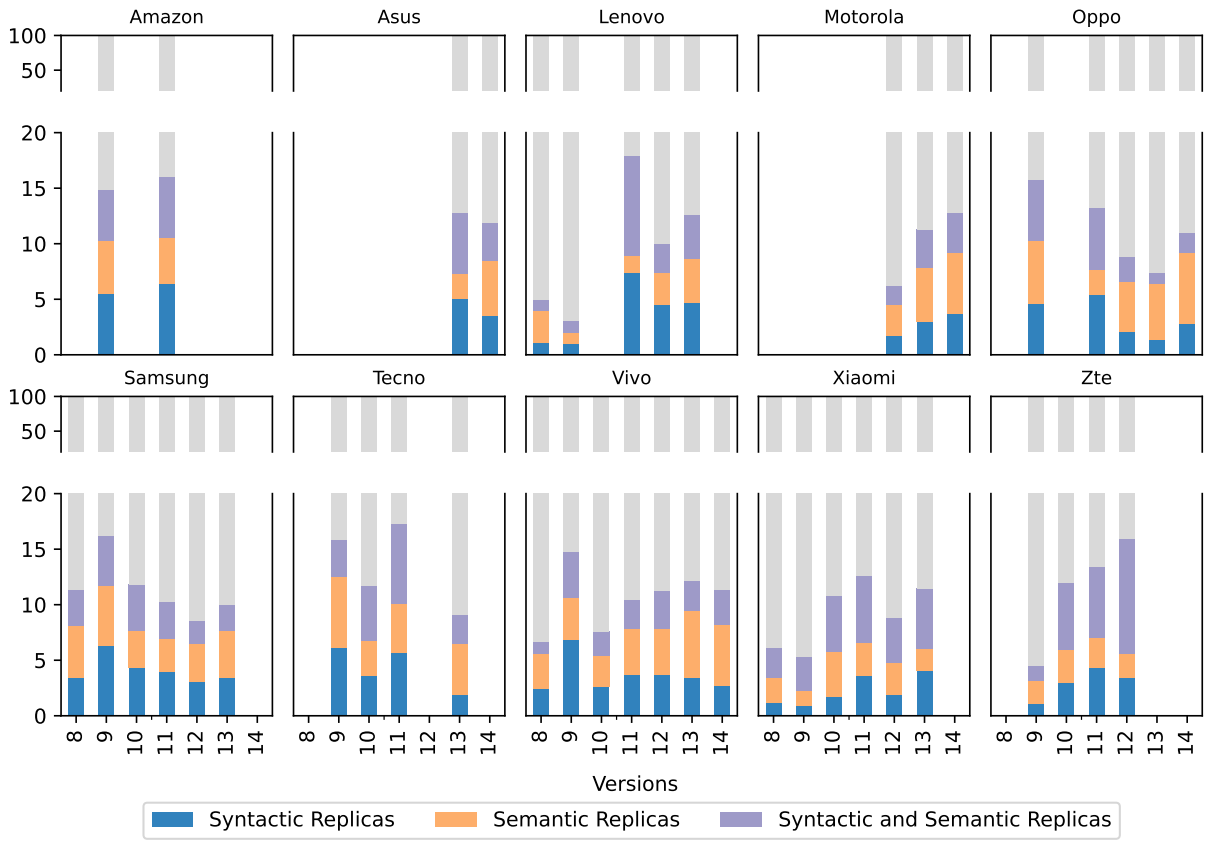

Figure 7.4: Replicas breakdown

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 7.4 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.

Syntactic vs Semantic Replicas. Figure 7.4 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).

Table 7.4: 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

Sensitivity to Similarity Threshold. We examine the sensitivity of RepFinder to variations in the similarity threshold used to detect Replicas (Section 7.5). 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 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 7.4, the threshold `0.85` achieves the best trade-off, leading to lower FPs and fewer missed true Replicas.

7.8.2 RQ2. Addition, Removal, and Inheritance

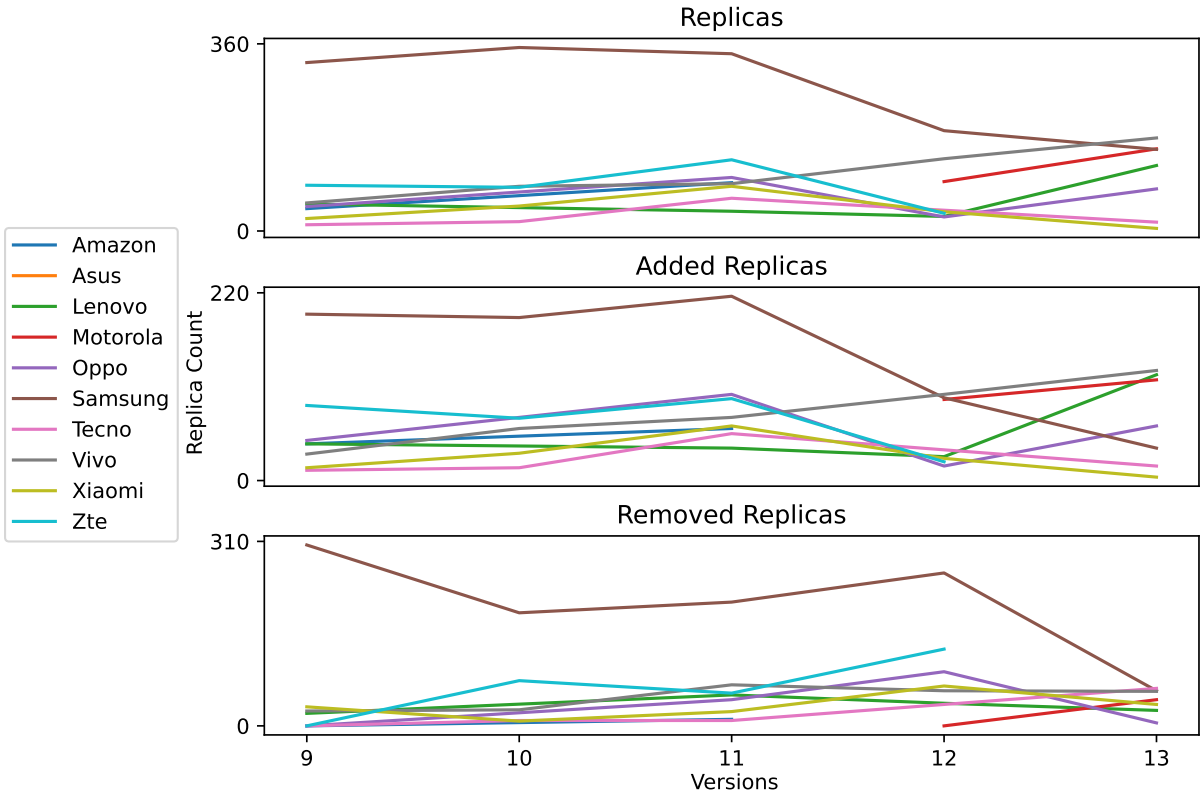

Figure 7.5: Addition and removal trends per vendor

As depicted in Figure 7.4, 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 7.5 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.

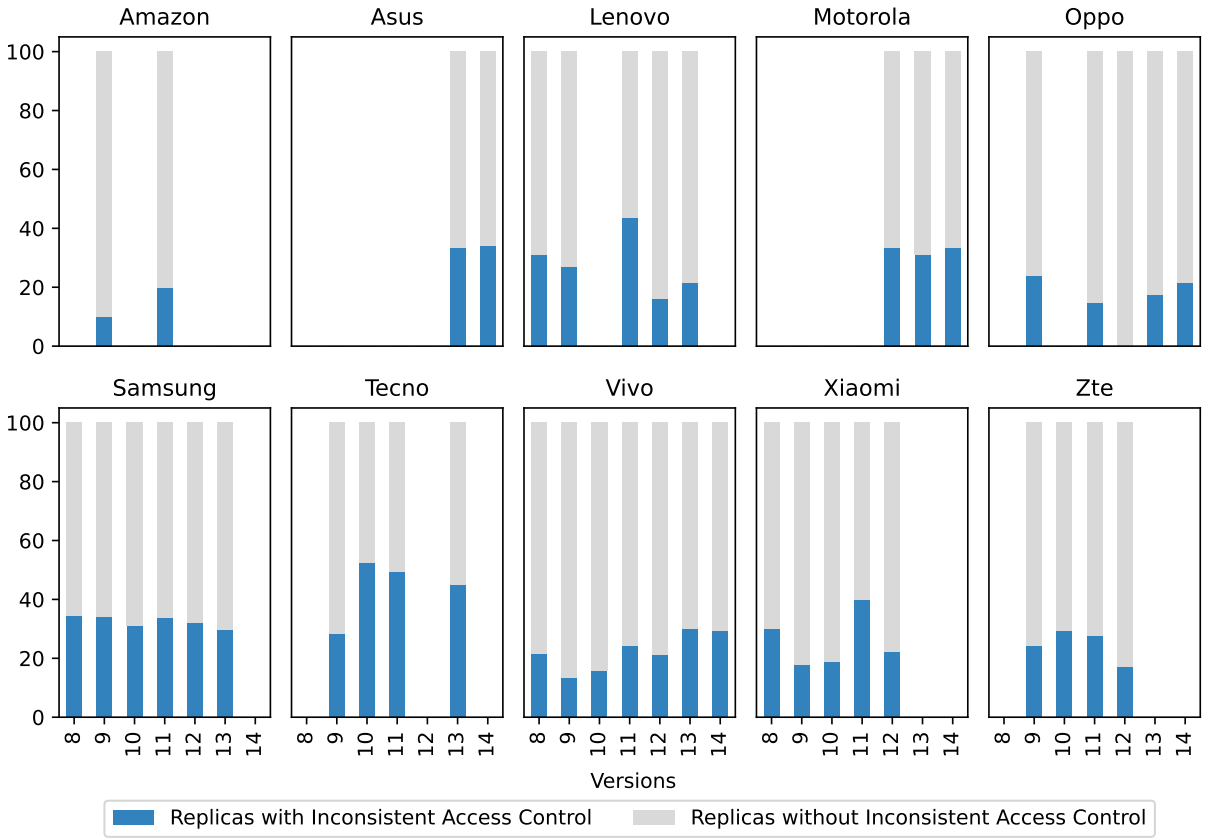

Figure 7.6: Access control inconsistencies in Replicas

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.

7.8.3 RQ3. Replicas Security Audit Results

Inconsistent Access Control. Figure 7.6 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

Table 7.5: Manually verified vulnerabilities

Device	Location	Impact	Status
ZTE-Axon 40	InputManager	KeyLogger	Ack
Vendor-A	PackageManager	Read Application directory	Ack
Vendor-A	BackupManager	Write to Application directory	Ack
Tecno-Spark	Telephony	Write to Secure System Settings	Ack
Tecno-Spark	Telephony	Read Build Serial Id	Ack
Vivo-X60	VivoPhoneLock	Read Sim Status	Reported
Vivo-X80	AudioService	Read Volume State	Reported

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 ratio (10%) of under-protected Replicas.

We refer the reader to Appendix C for a breakdown of security features behind the inconsistencies.

Security Updates Consistency. Here we devise the following experiment: For each <Original, Replica > API pair, we compare their access control implementation across subsequent versions (if both appear in two subsequent versions). If at least one of the APIs' security implementation changes, 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 one of the two APIs 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 7.7 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

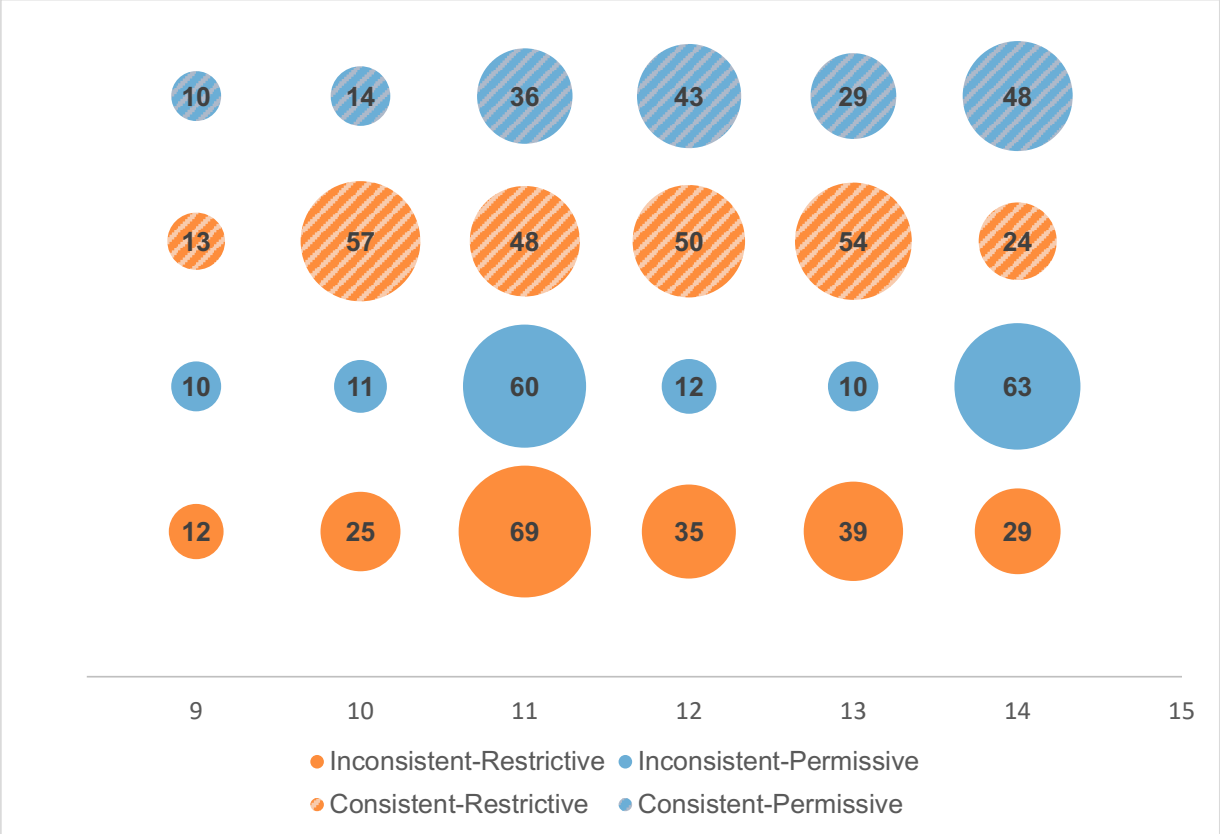

Figure 7.7: Security updates consistency in Replicas

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 7.5 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.

7.9 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 [136], 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[5].

We then extended our comparison to other open source tools that can handle Java clone detection. We obtained access to SourcererCC [106] and CCAligner[131], two state-of-the-art token-based code clone detectors. To allow comparison, we used JADX [113], to produce (approximate) Java source code of APIs from Android Dex file. We used the default threshold values specified in SourcererCC and CCAligner 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 CCAligner. In total, SourcererCC and CCAligner 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.

[5] as confirmed by the authors

Table 7.6: Clones detected by tested 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

Table 7.6 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.

Findings. Column #4 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 FPs. The main reason is that CCAligner looks for *large-gap clones* - that is, clones that reuse code with additions/subtractions of code lines. Android APIs (particularly those within the same service), often follow this pattern but they are not necessarily Replicas. This demonstrates that token-based code clone detection is not suitable for finding Replicas.

7.10 Threats to Validity

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, pinpointing 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.

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.

Concluding Part I

This part introduces three complementary techniques – **Bluebird**, **Ariadne**, and **RepFinder** – that systematically detect framework customization-induced access control vulnerabilities. Each technique leverages a distinct Android contextual feature:

- Bluebird captures application-level sensitivity cues,
- Ariadne models complex relations among data holders introduced through data-driven customization, and
- RepFinder relies on duplicate code in custom ROMs.

Individually, these contributions demonstrate how different dimensions of Android customization can introduce overlooked security weaknesses.

By bringing these complementary perspectives together, Part I provides a comprehensive answer to **RQ1**: *Android contextual features can be systematically leveraged to uncover framework customization-induced access control vulnerabilities missed by prior works.*

Part II: Adapting Access Control Analysis to Emerging Android Domains

The second research question (**RQ2**) asks how methods for uncovering access control vulnerabilities in Android can be adapted to emerging Android-based platforms, which introduce new architectural and contextual complexities. This part of the dissertation answers this research question using AAOS, a rapidly growing variant of Android that extends it into the automotive domain, as an application.

AAOS deviates from standard Android in several ways: its integration with vehicle hardware [58], its support for automotive-specific requirements, features, and technologies [50], and additional layers of OEM-driven customization [54]. These differences introduce new types of security risks that existing analyses, which are designed for Android on mobile devices, fail to capture.

To address these complexities, this part introduces **AutoAcRaptor**, a static analysis pipeline that systematically identifies access control and feature-check inconsistencies unique to AAOS. The following chapter details AutoAcRaptor and its evaluation on AAOS images. It demonstrates how Android analyses can be generalized and adapted to address the unique challenges of emerging Android-based platforms, thereby addressing **RQ2**.

Chapter 8

Auditing Access Control and Feature Checks in Android Automotive OS

This chapter is adapted from the paper [75], which investigates access control and feature-check anomalies in AAOS. This study shifts focus to *platform-specific adaptations*: the introduction of car services, feature gates, and automotive-specific entry points. It develops a static-analysis pipeline, AutoAcRaptor, to systematically identify and evaluate these AAOS-specific customizations, surfacing both access control inconsistencies and automotive feature-check anomalies. This material has been Reproduced with permission from Springer Nature.

8.1 Introduction

AAOS has rapidly gained prominence in the automotive industry as a fully integrated, auto-specific OS. Leading automakers, including Volvo, Polestar, and Stellantis, have already integrated AAOS into their vehicles, with others, such as Ford and Porsche, expected to follow [109].

Unlike Android Auto, which simply projects smartphone content on infotainment screens, AAOS is a base Android platform that extends Android[1] to support automotive functions such as entertainment, navigation, and system control. The extension, which spans various layers of the Android software stack, has been gradually carried out over 18 months before

[1]We refer to the unmodified version of the OS that does not support automotive functions as "Android" and "Traditional Android" interchangeably

AAOS was put into production. This design choice is concerning from a security perspective, given that Android has *evolved* to support AAOS rather than having it weighted-in from the beginning. Further complicating the situation is AAOS openness model, which offers automakers the flexibility to tailor the AAOS codebases to their needs, all without a central entity that oversees the customization process.

Although prior works [20, 68, 88, 93, 94, 95, 96] have studied some aspects of AAOS security, they are limited to investigating its external attack surface [94, 96] and specific enabling technologies [68, 93, 94]. *Evolution-related* vulnerabilities, which are likely to be introduced due to the AAOS design model and underregulated customization, remain understudied. For each new AAOS version and/or vehicle model, Google and automakers alter the latest AAOS codebase to embed new AAOS-specific and/or vendor-specific functional requirements. When such an alteration is not carried out properly, vulnerabilities are inevitable. Several prior works have investigated the impact of vendor customization in traditional Android setting (i.e., in mobile smartphones and tablets). However, to the best of our knowledge, none has looked at the security impact of customization in auto devices. To bridge the gap, we perform an empirical investigation of AAOS customization in a diverse set of AAOS ROMs. Our study aims to answer whether AAOS customization leads to potential anomalies, by analyzing ten AAOS images from four automakers. We focus our investigation on the framework layer, which implements AAOS APIs.

To facilitate the study, we develop a new automated analysis pipeline, AutoAcRaptor, which locates AAOS-related entry points and evaluates their security aspects. AutoAcRaptor performs static analysis of traditional and car-specific Android system services to identify entry points related to AAOS, that is, app-accessible APIs that are newly added or adapted for AAOS implementation. To ensure accurate identification, our tool relies on static patterns and naming indicators.

Evaluating the security aspects of AAOS entry points faces the following two main challenges: (1) due to the lack of functional specifications, it is unclear which Android functionality should be blocked or adjusted (e.g., disabling video playing when the car is in drive mode), and similarly (2) due to the lack of formal security documentation, it is unclear whether new APIs should implement access control (e.g., enforce a permission in a new API that enables accessing car properties). AutoAcRaptor meets these challenges by conducting consistency analysis of AAOS-feature and access control checks across AAOS APIs. Our intuition is that Google and automakers should implement consistent auto feature checks to limit, block access to, or adjust functionality of Android APIs. They should similarly implement consistent traditional security checks (e.g., car permissions) when exposing underlying functionality to apps. Failure to do so may allow unprivileged third-party apps to access unintended resources and even manipulate them without having

appropriate privileges. AutoAcRaptor leverages and adapts traditional consistency analysis techniques [1, 12, 15, 111] for our task. Particularly, AutoAcRaptor uses convergence and similarity analysis to identify functionally-similar APIs.

Measurement. We run AutoAcRaptor on ten AAOS ROMs; two correspond to the latest AOSP codebases (versions 13 and 14), and eight are customized by General Motors (GM), Honda, Volvo and Polestar. Our analysis identifies 34 car system services (i.e., new AAOS-specific services) and five modified system services (i.e., Android system services containing new or modified AAOS entry point), on average per ROM.

Furthermore, our study shows that Google has *modified* 75, 73 traditional entry points on versions 13, 14, respectively to accommodate automotive functionality. The extent of vendor customization varies across automakers; Honda (v12L) introduces only one new entry point, while GM-SUV-22 (V10) introduces up to 249 entries.

Security Analysis. AutoAcRaptor identifies eight entry points with an inconsistent feature check and nine with an inconsistent access control enforcement on average per ROM. The number varies across automakers, reaching up to 14 and 9 in AOSP AAOS v13. AOSP codebases include approximatiely 4% potential inconsistencies, which are inherited by all automaker codebases. From these inconsistencies, we identified ten potential vulnerabilities[2]. We have reported the cases to the corresponding automakers. At the time of writing, five have been acknowledged and the rest are pending verification.

Contributions. We make the following contributions:

- We conduct the first systematic investigation of AAOS framework.
- We design and implement AutoAcRaptor, a pipeline that combines static analysis and light NLP to generate and evaluate access control specification of AAOS-specific entry points at the framework layer.
- We evaluate our tool on ten AAOS ROMs, quantifying the extent of AAOS customization and demonstrating the existence of auto feature and access control anomalies.

8.2 Motivation

While AAOS has extended Android through new car-based, and customized system services, the process introduces inconsistencies that motivated our investigation. In particular,

[2]confirmed on automaker emulators.

```
public void setMasterMute(boolean mute) {
    if (!isPlatformAutomotive()) {
    // TODO: remove the isPlatformAutomotive check here.
    // isPlatformAutomotive check is for safety but may not be necessary.
        mute = false;
    }
    if ((isPlatformAutomotive() && userId==UserHandle.USER_SYSTEM)){
        mAudioSystem.setMasterMute(mute);
```

Figure 8.1: Uncertain automotive feature check in `AudioService`.

```
public int getMinLockLength(boolean isPin, int complexity) {
    complexity = PasswordMetrics.sanitizeComplexityLevel(complexity);
    // TODO: b/131755827 add devicePolicyManager support for Auto
    DevicePolicyManager dpM = mContext.getSystemService(Context.
        DEVICE_POLICY_SERVICE);
    PasswordMetrics adminMetrics =
    dpM.getPasswordMinimumMetrics(mContext.getUserId());
```

Figure 8.2: Intended future automotive support in `DevicePolicyService`.

developer comments and code evolution show uncertainty about which Android functionality should be blocked or adjusted, and whether new APIs should enforce feature or access control checks.

8.2.1 Motivating Examples of Developer Confusion

We observe that AAOS developers may not have accurate knowledge of which Android APIs or code pieces should be blocked or adjusted to integrate AAOS. Figure 8.1 illustrates an *uncertain* automotive feature check. As indicated in the developer's comment, the feature check is added for safety but may not be necessary. The check was marked "to be removed" in version 9 but still lingers in version 14.

In contrast, Figure 8.2 illustrates a "to be added" automotive feature check. Here, developers explicitly note that support for AAOS should be added in future versions, but the code remains incomplete. Such uncertainties complicate attempts to infer AAOS functional and security requirements from source code.

8.2.2 Motivating Examples of Inconsistent Checks

```
public void setLocationEnabled(boolean locationEnabled) {
    if (mIsAutomotive && !locationEnabled) {
        // Calls to disable location are ignored in  automotive builds
        return;
    }
    mInjector.binderWithCleanCallingIdentity(() -> {
       getLocationManager().setLocationEnabledForUser(locationEnabled);
    }
}
```

Figure 8.3: Simplified implementation of `DevicePolicyManager.setLocationEnabled`.

Beyond developer uncertainty, our analysis revealed cases where automotive feature or access control checks were inconsistently enforced. For example, in Figure 8.3, `DevicePolicyManager.setLocationEnabled` omits calls to `LocationManager.setLocationEnabledForUser` when the platform is automotive. However, `setLocationEnabledForUser` itself does not enforce the automotive feature check, allowing apps to bypass the intended restriction by invoking it directly. We reported this case to Google.

We refer the reader to Subsection 8.4.7 for additional examples of inconsistent access control checks discovered in our study.

8.3 System Design

In this section, we introduce a high-level overview of our tool, AutoAcRaptor, followed by an elaborate discussion of its key phases.

8.3.1 Approach Overview

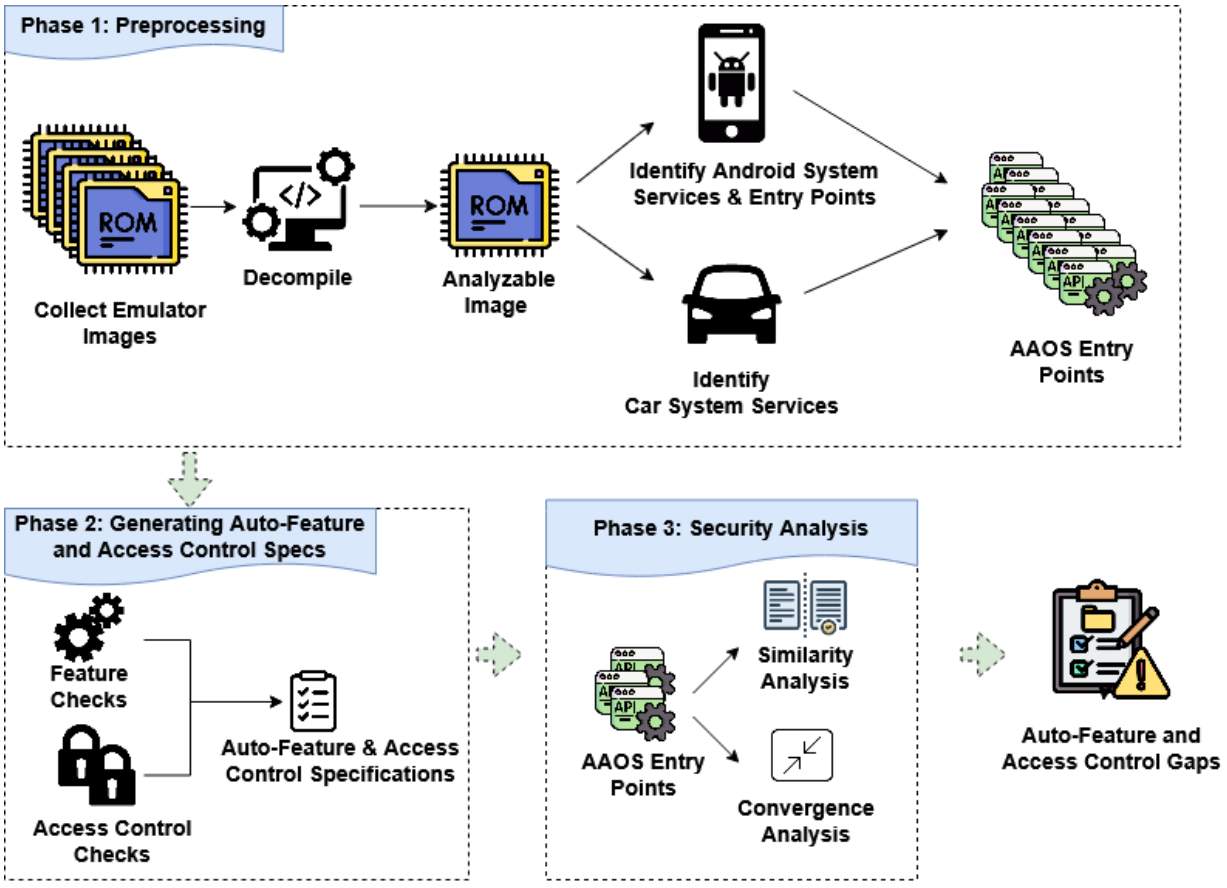

Figure 8.4: Approach overview of AutoAcRaptor

AutoAcRaptor consists of three phases, as depicted in Figure 8.4. In the first step, AutoAcRaptor preprocesses AAOS ROMs to extract framework classes. Since the analysis spans both Google and automaker AAOS ROMs[3], AutoAcRaptor operates on compiled framework bytecode. It then analyzes the framework classes to locate car system services, modified Android system services, and corresponding entry points. In the second phase, it analyzes the extracted entry points to generate a specification that maps resources (i.e., potential sinks in an entry point) to auto feature and access control checks. In the third

[3]Source code is unavailable for automaker (emulator) ROMs

phase, AutoAcRaptor leverages the specification to perform a ROM-wide security evaluation. Specifically, AutoAcRaptor groups *functionally-similar* entry points and compares their auto feature/ access control specification. Finally, it flags inconsistent cases as potential anomalies.

8.3.2 Phase 1: Preprocessing

In the first step, AutoAcRaptor decompiles target AAOS ROMs and prepares them for upcoming analyses. It extracts framework and car-specific libraries, using a variety of tools (apktool [123], dex2jar [101], smali [49], etc.).

Collecting AAOS images. Although the AAOS source code (Google version) is readily available in AOSP repositories [48], it is more difficult to obtain customized AAOS code. Specifically, (1) we cannot extract ROMs (for decompilation) from physical vehicles due to obvious cost and availability reasons, and (2) to the best of our knowledge, there are no public repositories for automaker source code. To address this challenge, we look for and collect available AAOS *emulator* images linked on the official Android Documentation webpage [30] and on some vendors' websites [89, 69, 98, 126]. Since these emulated images are provided to developers to test their car applications, they closely mimic the real ROMs deployed in cars[4]. More details on the collected images are discussed in Section 8.4.

Identifying car system services and entry points. Unlike Android system services, which are registered in the Service Manager, car system services are registered in a central system service *CarService*. Specifically, the implementation of *CarService*, i.e., the class `ICarImpl`, creates a new binder instance of each car system service, and adds it to a list of local services (via `CarLocalServices. addService` method) and to another list maintaining active services.

AutoAcRaptor leverages the above registration pattern to statically identify car system services as follows: it first locates the class implementing *CarService*'s Stub. It then builds a precise call graph of its constructor method (`init` method) and interprocedurally traverses it to identify invocations to `CarLocalServices. addService`. Finally, it retrieves the concrete type of the car system service by resolving the arguments of `addService`.

To identify app-accessible entry points, AutoAcRaptor locates the interface of each resolved car system service and extracts its defined public methods.

[4]Although we cannot guarantee that an emulated ROM is an exact copy of the actual ROM deployed in cars

```
boolean mFeature = false;
init(){
 if(mPMS.hasSystemFeature(PackageManager.FEATURE_AUTOMOTIVE) mFeature = true;
}
entryPoint(..){
 if(!mFeature) return;
}
```

Figure 8.5: Implicit Automotive feature checks

Identifying AAOS entry points in Android system services. AutoAcRaptor relies on the static patterns traditionally used by prior work to recover Android system services and their entry points [1, 111, 15, 64, 12](details are omitted for brevity). To pinpoint AAOS entry points in Android system services (i.e., entries modified/added to accommodate AAOS), AutoAcRaptor builds a control flow graph of each entry and inspects its conditional statements to locate those corresponding to automotive feature checks. Specifically, it leverages the following rules:

Rule 1: Direct automotive feature check. A conditional statement is classified as automotive if one operand in the predicate evaluates to an invocation of `PackageManager.hasSystemFeature` with the constant string *"android.hardware.type.automotive"* as an argument.

Rule 2: Implicit automotive feature check. AutoAcRaptor further tracks a special kind of control dependency that implies an automotive feature check. Consider the example in Figure 8.5: Variable `mFeature` does not directly indicate an automotive feature check. However, `mFeature = true` must imply that `mPMS.hasSystemFeature(PackageManager.FEATURE_AUTOMOTIVE)` equals to `true`. To handle such cases, AutoAcRaptor tracks control dependencies caused by such equivalence checks, which are prevalent in AAOS.

8.3.3 Phase 2: Generating Auto-Feature and Access Control Specs.

At this stage, AutoAcRaptor statically analyzes the extracted entry points to build a specification that maps *resources* to observed auto feature and access control checks.

Resource definition. A *resource* is defined as either (1) a reachable method invocation or field update instruction in an entry point or (2) the entry point itself.

Access control and auto feature checks. We perform a forward control-flow analysis on the API's interprocedural control flow graph (ICFG) and identify the conditional branches on which a resource is control dependent. A conditional branch is classified as an access control check if one of its operands corresponds to an invocation of known permission checks (e.g., `checkPermission`) or nonforgeable identifier checks (e.g., `UserHandle.getCallingUserId`). AutoAcRaptor similarly relies on Rule 1 (refer to Subsection 8.3.2) to identify auto feature checks. Note that unlike access control checks where we only consider instructions in the `true` branch, we process both the `true` and `false` branches of auto feature checks. As stated earlier, resources in the true branch are exclusive to AAOS, while those in the false branch are blocked on AAOS.

Resource reduction. As noted in prior work [35], not all resources control dependent on an access control are sensitive; consider a local variable update. We observe a similar phenomenon for auto feature checks; not all instructions guarded by an auto feature check are exclusive to AAOS. For example, a log statement is not relevant to AAOS even if it is control dependent on auto checks. We address this issue by relying on the following reduction strategies:

- **Frequency analysis:** AutoAcRaptor performs a frequency analysis to filter out commonly invoked resources across APIs, as they are less likely to be sensitive or AAOS-specific.
- **Removing data dependencies:** AutoAcRaptor conducts data flow analysis to identify and remove instructions exclusively used by the identified frequent instructions (e.g., `StringBuilder` methods used to construct log messages in `Log.d(..))`.
- **Resource name similarity analysis:** This strategy analyzes an API's resources to identify and exclude those that are less resembling the main functionality of the API through naming similarity analysis. Specifically, we use UniXcoder [67], a unified cross-modal pretrained NL-PL model for generating word embeddings. We use the model's tokenizer to tokenize the entry point name and a given resource name and generate corresponding embeddings. The latter are then compared using cosine similarity. If the calculated similarity is lower than an empirically selected threshold (see Section 8.4), the resource is not considered for further analysis.

The above strategies achieve a reduction of approximately 92% of instructions on average per entry point, narrowing down to the most relevant resources of an API.

$$
\begin{aligned}
AutoFeature &:= 1 \\
AccessControl_1 &:= [UserId = USER\_SYSTEM] \\
AccessControl_2 &:= [Perm = MODIFY\_AUDITO\_ROUTING] \\
AccessControl_3 &:= [Perm = INTERACT\_ACROSS\_USERS\_FULL] \\
AccessControl_4 &:= [Uid = SYSTEM]
\end{aligned}
$$

Figure 8.6: Generated specifications for `AudioService.setMasterMute`

Example of generated specifications. Figure 8.6 showcases the generated auto feature and access control specifications for the API resource `AudioService.setMasterMute(...)`. As the shown, the analysis identifies an auto feature check and four access control checks. [5]

8.3.4 Phase 3: Security Analysis.

To conduct the security analysis, AutoAcRaptor groups *functionally-similar* entry points and compares their auto feature and access control specification. Inconsistent specifications imply a potential anomaly. Recall that this solution addresses the lack of an oracle that can dictate whether a resource is *adequately* guarded by an auto-feature and/or access control check. AutoAcRaptor determines functional similarity through: Convergence and Similarity Analyses.

Convergence analysis: AutoAcRaptor conducts convergence analysis across entry points to identify those that overlap in functionality [111, 64]. This solution assumes that if the same resource is reachable using two entries, then they are similar. Note that, due to the large code size of AAOS frameworks, it is not trivial to identify converging entries. AutoAcRaptor achieves efficiency by using the specifications generated in Phase 2, which conveniently maps resources to observed auto feature and access control checks in different entries.

Similarity analysis: Unlike convergence analysis where an exact common resource is required to identify functionally-overlapping APIs, this method adopts a more relaxed assumption. It relies on similarity to find such APIs – for example, it considers APIs exhibiting similar names (e.g., `CarPropertyService. getProperty` and `CarPropertyService.getPropertyList`) to implement similar functionality. This analysis can cover APIs

[5] A similar specification is generated for each reachable resource within the API's implementation. The specs are omitted for brevity

missed by the convergence analysis. AutoAcRaptor uses UniXcoder to identify similar APIs, following the same method discussed in Phase 2 (in Resource Reduction).

8.4 Evaluation

We implement a prototype for AutoAcRaptor consisting of two components: (1) a static analysis component, built on top of WALA [103], and (2) a similarity analysis component, using UniXCoder. We evaluate AutoAcRaptor on a machine with Intel Core-i7 processor, featuring 12 cores with a maximum clock speed of 5.0 GHz, and 16GB RAM.

8.4.1 Target ROMs

We collected two AAOS ROMs from AOSP and ten automaker-customized AAOS ROMs. The AOSP ROMs span versions 13 and 14, while the custom ones span versions 9 to 14. The latter are customized by General Motors (GM), Honda, Volvo, and Polestar [89, 69, 126, 98].

When preprocessing the ROMs, we found that two ROMs, Volvo v10 and Polestar v9, could not be decompiled by apktool [123] or by smali [49]. Therefore, we excluded them from our analysis. Similarly, we found that another two ROMs, Honda v11 and GM-SUV-23 v11 were partially decompiled by the tools. However, we opted to include them in our analysis because we were able locate some AAOS system services and entry points. Table 8.1 lists the detailed statistics of the remaining ten ROMs (two AOSP and eight customized).

8.4.2 Research Questions

To evaluate our solution, we explore the following research questions:

- **RQ1:** What is the extent of Android customization carried out to accommodate AAOS?
- **RQ2:** What is the access control and auto-feature check landscape in AAOS entry points?
- **RQ3:** Does AAOS customization introduce access control and auto-feature check anomalies?
- **RQ4:** What is the accuracy of AutoAcRaptor?

8.4.3 RQ1: AAOS Customizations

Table 8.1: Detailed statistics of collected AAOS ROMs

(a) System services statistics

1. OS Image	2. Android Services	3. Car Services
GM-SUV-22 (V10)	165	31
Polestar 2 (V10)	142	30
GM-Cad-Lyriq-SUV-23 (V11)	159	30
GM-SUV-23 (V11)	133	30
Honda (V11)	124	29
Volvo-XC40 (V11)	151	29
GM-Cad-Lyriq-SUV-24 (V12L)	165	38
Honda (V12L)	163	36
AOSP (V13)	177	40
AOSP (V14)	168	44

(b) Entry points breakdown

1. OS Image	4. Total	Defining Service		AAOS Breakdown		
		5. Android Services	6. Car Services	7. Google Customized	8. Automaker Added	9. Automaker Modified
GM-SUV-22 (V10)	3103	2958	145	200	249	217
Polestar 2 (V10)	2859	2724	135	190	0	217
GM-Cad-Lyriq-SUV-23 (V11)	3237	3070	167	220	5	231
GM-SUV-23 (V11)	194	38	156	156	1	40
Honda (V11)	159	24	135	135	2	31
Volvo-XC40 (V11)	3213	3057	156	209	4	223
GM-Cad-Lyriq-SUV-24 (V12L)	2167	1908	259	309	20	38
Honda (V12L)	2133	1909	244	294	1	38
AOSP (V13)	3515	3254	261	336	-	-
AOSP (V14)	2210	1871	339	412	-	-

System Services. Columns 2 and 3 in Table 8.1 list Android and Car system services, respectively. On average, AutoAcRaptor identifies 155 Android system services and 34 Car system services. The number of Car system services reaches up to 38 in the custom images, specifically, in GM-Cad-Lyriq-SUV- 24 (V12L) and up to 44 in the AAOS AOSP images (in version 14).

Entry Points. Columns 4 in Table 8.1 lists the total number of entry points in the identified system services, while columns 5 and 6 break down the the entries by their defining class; Android system service (i.e., those defined also in non-AAOS builds), and Car system service. As shown, the number of entry points in car services varies from 135 in version 10 to 339 in version 14, exhibiting a steady increase throughout major versions.

The rest of the columns (columns 7-9) report a breakdown of AAOS entry points. Specifically, column 7 lists those entry points modified or added by Google for AAOS – this includes entries in both Android and Car system services; whereas the last two columns show the number of entries added and modified by automakers. We identify entry points modified by automakers by comparing against their reference AOSP ROM. For example, we compare Polestar 2 (V10) to AOSP V10. [6]

Note that entries in columns 7 and 9 might overlap, as an entry point can be modified by both Google and the automaker.

The number of modified/added APIs by Google for automotive support are largely consistent within a major version in the collected ROMs, with the exception of the two aforementioned partially corrupt ROMs (Honda v11 and GM-SUV-23 v11). We speculate that the slight differences are due to decompilation errors or due to minor version differences.

Automaker Customization Landscape. As shown in the last two columns, the number of added entries varies across vendors, ranging from none in Polestar 2, to 249 in GM-SUV-22. In addition, the automakers modify a high number of AOSP entries (defined in both Android and Car system services), ranging from 31 in Honda v11 and reaching up to 231 in GM-Cad-lyriq-SUV-23.

8.4.4 RQ2: Security Landscape

Table 8.2 reports the number of AAOS entry points that enforce an access control check (columns 2-5) and/or an auto-feature check (column 6). On average, around 60% of AAOS entry points enforce access control. As further shown, most added entries by automakers (column 4) do not enforce access control, while a larger portion of Google and automaker-modified entries (columns 3 and 5, respectively) enforces access control. Note that a lack of access control enforcement does not necessarily imply a security flaw when the underlying resources are not sensitive.

The last column lists entry points implementing auto feature checks. We observe that the checks are only present in AOSP and they are largely consistent across all ROMs.

[6] Note that we are not considering older AOSP ROMs (version < 13) in further analysis.

Table 8.2: Number of entry points with access control and auto-feature checks

OS Image	Access Control Checks				Auto-Feature Checks
	Car Services	Modified/Added by Google	Added by Automaker	Modified by Automaker	
GM-SUV-22 (V10)	73 (50.3%)	54 (27%)	33 (13.2%)	105 (48.3%)	47 (1.5%)
Polestar 2 (V10)	69 (51.1%)	54 (28.4%)	0 (0%)	105 (48.3%)	47 (1.6%)
GM-Cad-Lyriq-SUV-23 (V11)	94 (56.2%)	52 (23.6%)	1 (20%)	89 (38.5%)	44 (1.3%)
GM-SUV-23 (V11)	89 (57%)	0 (0%)	0 (0%)	17 (42.5%)	47 (24.2%)
Honda (V11)	85 (62.9%)	0 (0%)	0 (0%)	18 (58%)	47 (29.5%)
Volvo-XC40 (V11)	93 (59.6%)	52 (24.8%)	1 (25%)	98 (43.9%)	44 (1.3%)
GM-Cad-Lyriq-SUV-24 (V12L)	145 (55.9%)	50 (16.1%)	4 (20%)	28 (73.6%)	43 (1.9%)
Honda (V12L)	141 (57.7%)	50 (17%)	0 (0%)	29 (76.3%)	43 (2%)
AOSP (V13)	155 (59.3%)	72 (21.4%)	–	–	68 (1.9%)
AOSP (V14)	211 (62.2%)	72 (17.4%)	–	–	68 (3%)

8.4.5 RQ3: Access Control & Auto-Feature Check Anomalies

AutoAcRaptor uses two strategies to identify access control and auto-feature check anomalies:

1) Convergence Analysis. Column 2 in Table 8.3 reports the number of API pairs detected by AutoAcRaptor as converging on at least one common resource. Observe that the analysis only considers pairs where at least one API is automotive (i.e., an API that (1) enforces an automotive feature check, (2) is defined in Car services, or (3) is modified/added by an automaker).

Column 3 in Table 8.3 lists the number of pairs with an access control or feature check anomaly. As mentioned earlier, this corresponds to the cases where two converging APIs enforce different access control (e.g., one enforces a permission while the other does not), and/or only one implements an auto feature check. On average, AutoAcRaptor reports 144

Table 8.3: Convergence analysis results

	Number of API Pairs	
OS Image	**Total**	**Access Control & Auto-Feature check Anomalies**
GM-SUV-22 (V10)	163	9
Polestar 2 (V10)	160	9
GM-Cad-Lyriq-SUV-23 (V11)	135	14
GM-SUV-23 (V11)	39	0
Honda (V11)	38	16
Volvo-XC40 (V11)	134	15
GM-Cad-Lyriq-SUV-24 (V12L)	183	12
Honda (V12L)	177	0
AOSP (V13)	197	9
AOSP (V14)	221	8

unique pairs with overlapping resources per image, among which about 9% are flagged as anomalous. Such inconsistencies signal potential security flaws and warrant further investigation.

2) API-Name Similarity Analysis. We count the number of API pairs that share semantically-similar names. Again, AutoAcRaptor ensures that at least one API in each pair is automotive. The second column of Table 8.4 reports the count of such APIs whereas the third reports those with different access control or auto feature checks. On average, AutoAcRaptor reports that 55 pairs are similar, and 12% are flagged as potentially anomalous. Note that, the number of anomalies increase linearly with the increase in total number of similar APIs.

8.4.6 RQ4: AutoAcRaptor Accuracy

We evaluate AutoAcRaptor's accuracy based on its true positive (TP) and false positive (FP) rates. Due to the lack of ground truth, we resort to manual analysis to determine the positives of AutoAcRaptor. We consider a reported anomaly TP, if the pair of APIs perform the same sensitive functionality (identified through manual analysis) but enforces different access control and/or auto-feature checks. Conversely, we consider a reported anomaly as

Table 8.4: Similarity analysis results

	Number of API Pairs	
OS Image	**Total**	**Access Control & Auto-Feature Check Anomalies**
GM-SUV-22 (V10)	46	1
Polestar 2 (V10)	45	1
GM-Cad-Lyriq-SUV-23 (V11)	45	5
GM-SUV-23 (V11)	39	4
Honda (V11)	30	4
Volvo-XC40 (V11)	44	5
GM-Cad-Lyriq-SUV-24 (V12L)	62	8
Honda (V12L)	62	8
AOSP (V13)	70	10
AOSP (V14)	112	30

FP if the API pair with inconsistent checks actually perform *dissimilar functionality*, hence the inconsistency is reasonable. We perform this assessment on a representative ROM: GM-Cad-Lyriq-SUV-24 (V12L), and manually validate all reported cases in the convergence and similarity analyses.

As discussed in Section 8.3, AutoAcRaptor's performance is highly dependent on the similarity score by UniXCoder. Thus, AutoAcRaptor's TPs and FPs are dependent on the threshold chosen for the similarity score. Table 8.5 shows AutoAcRaptor's TP and FP rates under multiple threshold values, varying from 0.6 to 0.9. As shown in the table, the threshold 0.8 achieves the best trade-off, leading to lower FPs and higher TPs, thus we select this value in our analysis. AutoAcRaptor achieves a TP rate of about 71% and a FP rate of about 29% in the convergence analysis and achieves a TP rate of about 75% and a FP rate of about 25% in the similarity analysis. We manually analyzed about 10% of the randomly selected false positives and found the primary root causes: 1) for convergence analysis, AutoAcRaptor correctly identified a common convergence point but it is not the primary functionality of the APIs, and 2) for similarity analysis, the APIs are identified as similar based on their names but have no similarity in their behavior, that is, their functionality is significantly different.

To evaluate false negatives, we analyzed about 5% of system services (10 in total), and corresponding entry points that were not reported as converging or similar by AutoAcRaptor. The analysis, which spanned unique 7140 API pairs, took 7.5 working days by one author. We did not find any pair that warranted further inspection (i.e., contained at least one

Table 8.5: Impact of various similarity thresholds

	Convergence		Similarity
Threshold	**TP**	**FP**	**TP**
0.6	0.67	0.33	0.39
0.7	0.71	0.29	0.41
0.8	0.71	0.29	0.75
0.9	0.71	0.29	0.71

convergence point, or was similar), therefore we concluded that AutoAcRaptor 's false negative rate is negligible.

8.4.7 Case Studies of Observed Anomalies

To demonstrate the security impact of our tool, we randomly picked a few reported access control and auto feature check inconsistencies and attempted to build end-to-end PoCs. We note though that not all gaps are exploitable. The reasons are twofold. First, triggering an inconsistency may require certain conditions that are hard to meet (e.g., the API requires a hardware feature not available on the emulator). Second, the inconsistency lies in proprietary functionality that is difficult to analyze and to understand its execution output.

Nonetheless, we were able to confirm ten cases on corresponding AAOS emulators. All have been reported to the respective vendors. Table 8.6 summarizes the anomalies. Next, we discuss three representative cases:

Case 1: The `CarPowerManagementService` in GM-Cad-Lyriq-SUV-24 (V12L) defines two APIs: `finished` and `unregisterListener`. The former ensures that the caller has `(Car.PERMISSION_CAR_POWER)`, and verifies that the method is being called from either a system or the same process. If checks are successful, an internal method `finishedImpl` is invoked, which our tool flags as a main *resource*. The latter API, `unregisterListener`, performs a single check to ensure that the caller has `(Car.PERMISSION_CAR_POWER)`, and transitively invokes the same resource `finishedImpl`. It thus misses the process checks. We reported the case to the respective automaker (General Motors), who has acknowledged it and promised to address it in their next release.

Case 2: `startNattKeepalive` and `startNattKeepaliveWithFd` are two APIs from `ConnectivityService` in AOSP (V14). Both invoke `startNattKeepalive` internal method. However, the former requires a signature-level permission `(PACKET_KEEPALIVE_OFFLOAD)`,

Table 8.6: Summary of verified anomalies

OS Image	Entry Point	Observed Anomaly
AOSP (V13)	CarPropertyService.getProperty	Misplaced permission check
GM-Cad-Lyriq-SUV-24 (V12L)	CarPropertyService.registerListener	Missing permission check
AOSP (V13)	DevicePolicyManagerService. setPasswordMinimumLowerCase	Missing auto feature check
GM-SUV-22 (V10)	PhoneProjectionBinderService. isProjectionForeground	Missing permission check
GM-Cad-Lyriq-SUV-24 (V12L)	CarPowerManagementService. unregisterListener	Missing access control check
Honda (V12L)	AccountManagerService. getPreviousName	Missing access control check
AOSP (V14)	ConnectivityService. startNattKeepaliveWithFd	Missing permission check
AOSP (V14)	PowerManagerService.goToSleep	Missing auto feature check
AOSP (V14)	UserManagerService.setuserRestriction	Missing auto feature check
AOSP (V14)	LocationManagerService. setLocationEnabledForUser	Missing auto feature check

whereas the latter lacks this permission check, resulting in an inconsistency. We reported the case to the Google Android team.

```
1 public CarPropertyValue getProperty(int prop, int zone) {
2     if (mConfigs.get(prop) == null) {
3         Slogf.e(TAG, "propId not in config list:0x" + toHexString(prop));
4         return null;
5     }
6     CarServiceUtils.assertPermission(mContext, mHal.getReadPermission(prop));
7     return mHal.getProperty(prop, zone);
```

Figure 8.7: Partial bypass of access control check guarding car property IDs

Case 3: Another case that we observed was in a car-specific API. Figure 8.7 shows a partial bypass of the access control check guarding car property IDs in Line 3. Since the access control is placed after the log operation, it discloses the car configuration data - as the log message is dumped within the calling app's process scope. In other APIs of this service, this information is protected. We reported it to Google, who promised to remediate it in future versions.

Concluding Part II

This part addresses the second research question (**RQ2**) by adapting access control analysis to AAOS. Through **AutoAcRaptor**, we show that Android-focused techniques can be extended by:

- **Modeling AAOS-specific entry points and car services**, to cover interfaces absent in phone-centric Android.
- **Auditing both access control and feature checks**, to capture inconsistencies.

Applied to multiple vendor AAOS ROMs, AutoAcRaptor reveals recurring gaps and acknowledged issues, demonstrating the practicality of adapting Android analyses to new domains.

Overall, these results provide a *partial answer* to **RQ2**: context-aware access control analysis can be extended to AAOS, but a comprehensive answer requires exploring other Android-based platforms such as WearOS [62] and Android XR [61], each with their own security considerations.

Chapter 9

Discussion

This chapter examines how the techniques introduced in this dissertation can be integrated into a unified framework and highlights the challenges involved in such integration. While Bluebird, Ariadne, RepFinder, and AutoAcRaptor each target different types of access control inconsistencies, they all rely on contextual signals—sensitivity cues in apps, data-driven customizations, semantic replication, or platform-specific feature checks—that can be combined to build a more holistic pipeline.

9.1 Toward a Unified Solution

While the app-side hints allow Bluebird to uncover novel access control gaps, it also limits its coverage to APIs that have app-side invocations. Ariadne and RepFinder suffer from a similar limitation due to their reliance on data-driven hints and existence of code replication respectively. Since this limitation stems from the narrow scope of each individual tool, it can be alleviated by combining them, and integrating other complementary prior works, to create a more holistic approach. Moreover, since these tools rely on the existence of hints, they can significantly benefit from the additional cues complementary techniques can provide. For example, an API with only a few weak data-driven, and a few weak app-side sensitivity indicators will benefit from a combination of both, which in conjunction can provide a strong access control recommendation. Adding RepFinder can further aid the overall detection of vulnerabilities and recommendation of access control by providing yet another class of signals that, when integrated, increases accuracy and coverage.

Incorporating existing probabilistic and convergence-based tools such as Poirot [35], Kratos [111], and ACMiner [64] can significantly enhance this unified pipeline by: (1)

increasing coverage through the inclusion of additional types of vulnerabilities, and (2) by reducing false positives, since tools like Poirot can accurately correlate access control checks with their intended target code instructions, the lack of which is the primary source of false positives in the proposed tools.

Finally, AutoAcRaptor can be integrated to ensure that the overarching solution also covers automotive-specific services and feature checks, while reducing its own false positives stemming from the usage of simplistic static analysis techniques like reachability.

9.2 Integration Challenges

While a unified solution holds promise, its realization is non-trivial. The first challenge is the heterogeneity in the output of the tools. For instance, Bluebird outputs a simple boolean recommendation of whether an API is under-protected, due to its reliance on binary classification of UI-based sensitivity indicators. Ariadne, in contrast, not only flags inconsistencies but also recommends specific missing access control. To integrate the two, Bluebird would need to be extended with a new classifier that maps UI-based sensitivity indicators to concrete traditional access control. Designing such a classifier is non-trivial, requiring novel methods that can bridge qualitative UI cues to access control checks. Similarly, integrating RepFinder poses challenges because it only identifies code-path replicas and flags inconsistencies among them. To be useful in a combined pipeline, these inconsistencies would need to be translated into actionable hints that directly support access control recommendations. Developing this translation layer is an open research problem.

More broadly, an overarching solution must reconcile heterogeneous evidence, prevent duplication of results, and balance breadth of coverage with precision, all while scaling to the size and complexity of real-world Android codebases.

9.3 Broader Outlook

Ariadne, Bluebird, and RepFinder illustrate that contextual features provide a powerful lens for uncovering new classes of vulnerabilities in Android, by each leveraging a different feature to expose a new class of inconsistencies. While this comprehensively answers RQ1, it does not exhaust the space of contextual features. Future research should explore methods to systematically identify additional features (potentially in an automated fashion) and design integration frameworks that simplify incorporating new analyses into a unified solution.

RQ2, on the other hand, has only been partially addressed. AutoAcRaptor demonstrates that existing techniques can be adapted to analyze AAOS, but many other Android-based platforms such as Android TV [3], Wear OS [62], Android Go [63], and Android XR [61] remain unexplored. Each introduces distinct architectural and platform constraints that may demand new adaptations. Whether the same techniques can generalize across all these variants, and how they must be extended to do so, is an open question requiring dedicated investigation.

Chapter 10

Conclusion

Building on the methods and insights developed throughout this dissertation, future research can explore new directions that push the boundaries of access control analysis in Android and its variants by integrating richer contextual signals (e.g., multimodal UI cues), combining complementary analysis techniques, and incorporating modern semantic models.

In sum, this dissertation contributes both methods and insights toward principled, scalable, and adaptable access control analysis in complex software ecosystems, and outlines a roadmap for advancing security in an increasingly diverse Android landscape.

References

[1] Yousra Aafer, JianJun Huang, Yi Sun, Xiangyu Zhang 0001, 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*, 2018.

[2] 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.

[3] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android SmartTVs vulnerability discovery via Log-Guided fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2759–2776. USENIX Association, August 2021.

[4] 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.

[5] Hamed Amini, Moez Draief, and Marc Lelarge. Flooding in weighted random graphs. In *2011 Proceedings of the Eighth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 1–15. SIAM, 2011.

[6] Anestisb. vdexextractor: Tool to decompile & extract android dex bytecode from vdex files. https://github.com/anestisb/vdexExtractor, 2024.

[7] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, page 794–807, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386026.

[8] AOSP. Lpunpack and lpmake. https://github.com/LonelyFool/lpunpack_and_lpmake, 2024.

[9] Aptoide. Aptoide, Accessed on: November 3, 2023. https://en.aptoide.com/.

[10] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[13] Yann Bachy, Frédéric Basse, Vincent Nicomette, Eric Alata, Mohamed Kaâniche, Jean-Christophe Courrège, and Pierre Lukjanenko. Smart-TV security analysis: Practical experiments. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 497–504, 2015. doi:10.1109/DSN.2015.41.

[14] 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. doi:10.1145/2976749.2978333.

[15] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the 25th USENIX Security Symposium*, page 48. USENIX Association, 2016.

[16] Anestis Bechtsoudis. simg2img. https://formulae.brew.sh/formula/simg2img, 2024.

[17] 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.

[18] Eduardo Blázquez, Sergio Pastrana, Álvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. Trouble over-the-air: An analysis of fota apps in the android ecosystem. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1606–1622. IEEE, 2021.

[19] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[20] Efstratios Chatzoglou, Georgios Kambourakis, and Vasileios Kouliaridis. A multi-tier security analysis of official car management apps for android. *Future Internet*, 13(3):58, 2021.

[21] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. https://www.usenix.org/conference/usenix-security-11/comprehensive-experimental-analyses-automotive-attack-surfaces, 2011. Accessed: 2023-12-20.

[22] C. J. CLOPPER and E. S. PEARSON. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 12 1934. arXiv:https://academic.oup.com/biomet/article-pdf/26/4/404/823407/26-4-404.pdf, doi:10.1093/biomet/26.4.404.

[23] OpenAI Codex. Openai codex. https://openai.com/blog/openai-codex, 2024.

[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] cs.android.com. Car-lib. https://cs.android.com/android/platform/superproject/main/+/main:packages/services/Car/car-lib/, n.d. Accessed: 2023-12-20.

[26] George Dantzig and Delbert Ray Fulkerson. On the max flow min cut theorem of networks. *Linear inequalities and related systems*, 38:225–231, 2003.

[27] 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. arXiv:2306.06347.

[28] Abdallah Dawoud and Sven Bugiel. Bringing balance to the force: Dynamic analysis of the android application framework. In *Network and Distributed Systems Security (NDSS) Symposium 2021*, February 2021.

[29] Luc De Raedt, Angelika Kimmig, Hannu Toivonen, and M Veloso. ProbLog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007.

[30] Android Developers. Test using the android automotive os emulator. https://developer.android.com/training/cars/testing/emulator, n.d. Accessed: 2023-12-20.

[31] Laura Dietz, Valentin Dallmeier, Andreas Zeller, and Tobias Scheffer. Localizing bugs in program executions with graphical models. *Advances in Neural Information Processing Systems*, 22, 2009.

[32] 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. doi:10.1145/3597503.3608140.

[33] Firmware Drive. https://firmwaredrive.com/index.php, 2021.

[34] 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. doi:10.1145/3460120.3485374.

[35] 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*, CCS '22, page 937–950, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560710.

[36] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2379–2396, 2020.

[37] Hugging Face. Bert cased model, Accessed on: February 8, 2023. https://huggingface.co/bert-base-cased.

[38] Faten Fakhfakh, Mohamed Tounsi, and Mohamed Mosbah. Cybersecurity attacks on CAN bus based vehicles: a review and open challenges. *Library hi tech*, 40(5):1179–1203, 2022.

[39] 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. doi:10.1109/TIFS.2017.2687880.

[40] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[41] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[42] FirmwareDrive. Firmware drive, Accessed on: December 8, 2021. https://firmwaredrive.com/.

[43] 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. doi:10.1109/SP.2017.31.

[44] GitLab. Android dumps · gitlab. https://dumps.tadiphone.dev/dumps, n.d. Accessed: 2023-12-20.

[45] 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.

[46] Google. Google Developers platform architecture. https://developer.android.com/guide/platform, 2022.

[47] Google. Android images. https://developers.google.com/android/ota, 2024.

[48] Google. Git repositories on android. https://android.googlesource.com/, 2024.

[49] Google. smali. https://github.com/google/smali, 2024.

[50] Google. Android automotive & android. https://source.android.com/docs/automotive/start/what_automotive#android-automotive-android, 2025.

[51] Google. Android discretionary access control (dac), 2025. https://source.android.com/docs/core/permissions/filesystem.

[52] Google. Android manifest, 2025. https://developer.android.com/guide/topics/manifest/manifest-intro.

[53] Google. Binder overview, 2025. https://source.android.com/docs/core/architecture/ipc/binder-overview.

[54] Google. Oem android system services. https://source.android.com/docs/automotive/custom_input#system-services, 2025.

[55] Google. Permission protection levels, 2025. https://developer.android.com/guide/topics/manifest/permission-element#plevel.

[56] Google. Permissions on android, 2025. https://developer.android.com/guide/topics/permissions/overview.

[57] Google. Selinux in android, 2025. https://source.android.com/docs/security/features/selinux.

[58] Google. Vehicle hal. https://source.android.com/docs/automotive/vhal, 2025.

[59] Google. Factory images for nexus and pixel devices, Accessed on: December 8, 2021. https://developers.google.com/android/images.

[60] Google. Google play store, Accessed on: November 3, 2023. https://play.google.com/store/.

[61] Google. Android xr, Accessed on: September 06, 2025. https://www.android.com/intl/en_ca/xr/.

[62] Google. Wear os, Accessed on: September 06, 2025. https://wearos.google.com/intl/en_ca/.

[63] Google. Android go, Accessed on: September 26, 2025. https://www.android.com/versions/go-edition/.

[64] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. *ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware*, page 25–36. Association for Computing Machinery, New York, NY, USA, 2019.

[65] Thomas Griffiths and Alan Yuille. A primer on probabilistic inference. *The probabilistic mind: Prospects for Bayesian cognitive science*, pages 33–57, 2008.

[66] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[67] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[68] B. Gözübüyük, B. Tang, K.G. Shin, and M.D. Pesé. Analyzing privacy implications of data collection in android automotive os. *arXiv.org*, September 2024.

[69] Honda. Honda android automotive os emulator. https://global.honda/en/cars-apps/index.html, 2024.

[70] 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.

[71] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. SUPOR: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 977–992, 2015.

[72] 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.

[73] 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.

[74] JesusFreke. Smali: Smali/baksmali. https://github.com/JesusFreke/smali, 2024.

[75] Jumana, Parjanya Vyas, and Yousra Aafer. Red light for security: Uncovering auto feature check and access control gaps in aaos. In Manuel Egele, Veelasha Moonsamy, Daniel Gruss, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 147–166, Cham, 2025. Springer Nature Switzerland. doi:10.1007/978-3-031-97623-0_9.

[76] 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. doi:10.1109/TSE.2002.1019480.

[77] Sayali Kate, John-Paul Ore, Xiangyu Zhang, Sebastian Elbaum, and Zhaogui Xu. Phys: Probabilistic physical unit assignment and inconsistency detection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 563–573, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236024.3236035.

[78] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001. doi:10.1109/18.910572.

[79] KU Leuven. Problog, 2022. https://dtai.cs.kuleuven.be/problog/.

[80] Jonathan Levin. Imgtool. https://newandroidbook.com/tools/imjtool.html, 2024.

[81] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using s park. In *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12*, pages 153–169. Springer, 2003.

[82] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th*

IEEE International Conference on Software Engineering, volume 1, pages 280–291. IEEE, 2015.

[83] 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.

[84] Rui Li, Wenrui Diao, Zhou Li, Jianqi Du, and Shanqing Guo. Android custom permissions demystified: From privilege escalation to design shortcomings. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 70–86, 2021. doi:10.1109/SP40001.2021.00070.

[85] Renju Liu and Felix Xiaozhu Lin. Understanding the characteristics of android wear os. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 151–164, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2906388.2906398.

[86] Yonghui Liu, Li Li, Pingfan Kong, Xiaoyu Sun, and Tegawendé F. Bissyandé. A first look at security risks of android TV apps. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 59–64, 2021. doi:10.1109/ASEW52652.2021.00023.

[87] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, 2012.

[88] Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy. A security analysis of an in-vehicle infotainment and app platform. In *10th USENIX workshop on offensive technologies (WOOT 16)*, 2016.

[89] General Motors. Gm developers. https://developer.gm.com/in-vehicle-apps, 2024.

[90] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 993–1008, 2015.

[91] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. *SIGPLAN Not.*, 51(1):469–484, jan 2016. doi:10.1145/2914770.2837661.

[92] 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.

[93] Mert D Pesé, Jay W Schauer, Murali Mohan, Cassandra Joseph, Kang G Shin, and John Moore. PRICAR: Privacy framework for vehicular data sharing with third parties. In *2023 IEEE Secure Development Conference (SecDev)*, pages 184–195. IEEE, 2023.

[94] Mert D Pesé and Kang G Shin. Survey of automotive privacy regulations and privacy-related attacks. 2019.

[95] M. Pesé, K. Shin, J. Bruner, and A. Chu. Security analysis of android automotive. *SAE Technical Paper*, 2020-01-1295, 2020. doi:10.4271/2020-01-1295.

[96] M.D. Pesé. A first look at android automotive privacy. *SAE Technical Paper*, 2023-01-0037, 2023. doi:10.4271/2023-01-0037.

[97] Michal Pióro and Deep Medhi. *Routing, flow, and capacity design in communication and computer networks*. Elsevier, 2004.

[98] Polestar. Polestar developer portal. https://www.polestar.com/global/developer#emulator, 2024.

[99] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In IEEE, editor, *S&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.

[100] Android Open Source Project. What is android automotive? https://source.android.com/docs/automotive/start/what_automotive, n.d. Accessed: 2025-09-23.

[101] pxb1988. dex2jar. https://github.com/pxb1988/dex2jar, 2024.

[102] Sampath Rajapaksha, Harsha Kalutarage, M Omar Al-Kadri, Garikayi Madzudzo, and Andrei V Petrovski. Keep the moving vehicle secure: Context-aware intrusion detection

system for in-vehicle CAN bus security. In *2022 14th International Conference on Cyber Conflict: Keep Moving!(CyCon)*, volume 700, pages 309–330. IEEE, 2022.

[103] WALA GitHub Repository. Wala github repository. https://github.com/wala/WALA, n.d. Accessed: 2023-12-20.

[104] R Tyrell Rockafellar. *Network flows and monotropic optimization*, volume 9. Athena scientific, 1999.

[105] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 43–55, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/504282.504286.

[106] 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. doi:10.1145/2884781.2884877.

[107] Samfrew. Original samsung firmware updates. https://samfrew.com/, 2019.

[108] SamMobile. Home - sammobile. https://www.sammobile.com/, June 2022.

[109] SamMobile. Porsche cars to feature android automotive, google play store integration soon. https://www.sammobile.com/news/porsche-cars-android-automotive-google-play-store-soon/, 2023. Accessed: 2023-12-20.

[110] Samsung. Samsung galaxy app store, Accessed on: November 3, 2023. https://www.samsung.com/us/apps/galaxy-store/.

[111] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[112] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. Internet Society, 5 2017. doi:10.14722/ndss.2016.23046.

[113] Skylot. jadx: Dex to java decompiler. https://github.com/skylot/jadx, 2024.

[114] Sampath Srinivas. A generalization of the noisy-or model. In *Uncertainty in artificial intelligence*, pages 208–215. Elsevier, 1993.

[115] StatCounter. Statcounter global stats 2024. https://gs.statcounter.com/vendor-market-share/mobile/worldwide/#monthly-201003-202007, 2024.

[116] steadfasterX. Steadfasterx/salt: Salt - [s]teadfasterx [a]ll-in-one [l]g [t]ool. https://github.com/steadfasterX/SALT, 2024.

[117] 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. doi:10.1145/3574158.

[118] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. AutoISES: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[119] Andrew S. Tanenbaum. *Computer Networks (5th edition)*. Pearson Education, 2010.

[120] 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.

[121] Marcos Tileria and Jorge Blasco. Watch over your tv: a security and privacy analysis of the android tv ecosystem. *Proceedings on Privacy Enhancing Technologies*, 2022.

[122] Marcos Tileria, Jorge Blasco, and Guillermo Suarez-Tangil. WearFlow: Expanding information flow analysis to companion apps in wear OS. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 63–75, San Sebastian, October 2020. USENIX Association.

[123] Connor Tumbleson. Apktool. https://apktool.org/, 2024.

[124] Güliz Tuncay, Soteris Demetriou, Karan Ganju, and Carl Gunter. Resolving the predicament of android custom permissions. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 01 2018. doi:10.14722/ndss.2018.23221.

[125] Dheeraj Vagavolu, Yousra Aafer, and Meiyappan Nagappan. (deep) learning of android access control recommendation from static execution paths. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*, pages 758–772, 2024. doi:10.1109/EuroSP60621.2024.00047.

[126] Volvo. Android emulator. https://developer.volvocars.com/in-car-apps/android-emulator-xc40/, 2024.

[127] Parjanya Vyas, Haseeb Ur Rehman Faheem, Yousra Aafer, and N Asokan. Ariadne: Navigating through the labyrinth of Data-Driven customization inconsistencies in android. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4245–4264, 2025.

[128] 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, Anaheim, CA, August 2023. USENIX Association.

[129] 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. doi:10.1109/ICSE48619.2023.00013.

[130] Min Wang, Pengcheng Wang, and Yun Xu. CCSharp: An efficient three-phase code clone detector using modified PDGs. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 100–109, 2017.

[131] 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.

[132] Qiyan Wang and Sanjay Sawhney. VeCure: A practical security framework to protect the can bus of vehicles. In *2014 International Conference on the Internet of Things (IOT)*, pages 13–18. IEEE, 2014.

[133] 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.

[134] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161*, 2020.

[135] 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. doi:10.1145/2508859.2516728.

[136] 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.

[137] 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.

[138] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. Iconintent: Automatic identification of sensitive ui widgets based on icon classification for android apps. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 257–268. IEEE Press, 2019. doi:10.1109/ICSE.2019.00041.

[139] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 607–618, 2016.

[140] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[141] Doguhan Yeke, Muhammad Ibrahim, Güliz Seray Tuncay, Habiba Farrukh, Abdullah Imran, Antonio Bianchi, and Z. Berkay Celik. Wear's my data? understanding the cross-device runtime permission model in wearables. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2404–2421, 2024. doi:10.1109/SP54263.2024.00077.

[142] 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. arXiv:2102.09764.

[143] Syeda Mashal Abbas Zaidi, Shahpar Khan, Parjanya Vyas, and Yousra Aafer. A longitudinal analysis of replicas in the wild wild android. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1821–1833, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3691620.3695546.

[144] 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. arXiv:2102.08172.

[145] 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.

[146] 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.

[147] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating insecure input validations in android services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1165–1178, 2018.

[148] Mu Zhang and Heng Yin. Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[149] 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.

[150] 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.

[151] 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.

[152] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. In *The 2022 Network and Distributed System Security Symposium (NDSS'22)*, 2022.

[153] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, page 185–196, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2435349.2435377.

[154] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423, 2014. doi:10.1109/SP.2014.33.

[155] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012. doi:10.1109/SP.2012.16.

[156] 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

Bluebird

A.1 Additional Case Studies of Access Control Gaps

Manipulating wifi settings. Bluebird reported an access control gap in Samsung's `WifiService. setWifiSharingEnabled`. As hinted by its name, the API allows changing custom Wifi Sharing settings. We manually investigated the case and found that the API's implementation enforces a normal-level permission `android.permission.ACCESS_WIFI_STATE`. We further checked its app-side constraints, generated by Bluebird in three preloaded apps (*SecSettings*, *SystemUI*, and *SystemUIDesktop*). The constraints all propagate a high sensitivity estimation from the app entries to the API, which we identified to be related to a protected broadcast action, a system-level permission, and a sensitive UI-dialog. Essentially, all constraints pointed to a gap in the API's enforced access control. Samsung acknowledged and confirmed that they have added a more adequate check for the API in later versions.

Manipulating Smart-suspend Setting. In Fire HD 10, our audit reported a non-compliant API `SmartSuspendManagerService.setScheduleType`, which controls a custom user preference *Smart Suspend Type*. The API does not enforce any access controls, resulting in a low prior probability. The audit result indicated a higher probability stemming from the sensitive UI based clues from *TabletSettings* app, which triggers the API in a high-sensitivity UI-block (sensitivity was inferred through a few indicators). We investigated the case and found that the proprietary setting allows to automatically suspend the device (which includes operations such tearing down connections).

Shutting down Bluetooth connections. In the same Samsung ROM, Bluebird identified another non-compliant API `IBluetoothManager.shutdown()` which tears down es-

tablished bluetooth connections. Our investigation reveals that the API was initially assigned a low-sensitivity prior probability, since it enforces a normal-level permission `android.permission.BLUETOOTH_ADMIN`. The posterior probability indicated a high sensitivity estimation via a few clues extracted from *Bluetooth* preloaded app. For example, the API was invoked in a background service that enforced a programmatic system permission `android.permission.BLUETOOTH_PRIVILEGED` – Observe the permission name similarity which has enabled generating a high contribution constraint.

A.2 ProbLog Rules for Table 5.1

```
% 'Api' denotes an audited framework API.
% 'Ep' denotes an identified app entry discovered through app analysis.
% 'Id' denotes a unique identifier assigned to each observation.

% Framework-side Evidences:

% Rule 1:
% The random variable 'api_ac' indicates the probability of 'Api' being sensitive.

% 'fw_no_ac' denotes the observation that an API enforces weak or no access control.
0.1 :: api_ac(Api) :- fw_no_ac(Api, Id).

% App-side Evidences:

% Rules 2 and 3
% The random variable 'ep_ac' indicates the probability of 'Ep' being sensitive.
% 'app_ep_ac' and 'app_ep_no_ac' denote the observations that 'Ep' enforces strong strong access control, or no
    access control, respectively.

0.9 :: ep_no_ac(Ep) :- app_ep_no_ac(Ep, Id).
1 :: \+ep_ac(Ep) :- ep_no_ac(Ep, Id).
0.9 :: ep_ac(Ep) :- app_ep_ac(Ep, Id).

% Rule 4
% 'app_alert_dialog' denotes that observation that 'Ep' is a sensitive UI type (e.g., alert dialog)
% 'ep_alert_dialog' denotes that the probability of 'Ep' being sensitive because it is a sensitive UI type.

0.9 :: ep_alert_dialog(Ep) :- app_alert_dialog(Ep, Id).
0.8 :: ep_ac(Ep) :- ep_alert_dialog(Ep)(Ep, Id).

% Rules 5, 6, and 7 (constructed similarly to Rule 4 above):
0.9 :: ep_non_cancellable_alert_dialog(Ep) :- app_non_cancellable_alert_dialog(Ep, Id).
0.8 :: ep_ac(Ep) :- ep_non_cancellable_alert_dialog(Ep, Id).


0.9 :: ep_multi_confirm(Ep) :- app_multi_confirm(Ep, Id).
0.8 :: ep_ac(Ep) :- app_multi_confirm(Ep, Id).

Score :: nlp_sensitivity(Ep) :- app_nlp_sensitivity(Ep, Score, Id).
0.9 :: ep_ac(Ep) :- nlp_sensitivity(Ep, Id).
```

```
40
41 % App-side Clues:
42
43 % Rule 8:
44 % 'nlp_contribution' denotes the observation that 'Api' is relevant to 'Ep' with a computed contribution score 'CT'
       (defined in Section 4.3)
45 % 'CT_p' denotes the (computed) implication probability of 'Api' being sensitive based on its contribution score to
      'Ep'; Specifically, CT_p = CTScore * 0.9
46
47 CT_p :: api_ac(Api) :- ep_ac(Ep), nlp_contribution(Ep, Api, CT, Id).
48
49 % Rules 9 to 12 (constructed similarly to the implication constraint depicted in Rule 8):
50 % Rule 9:
51 % 'modifiable_context' denotes the observation that the context of 'Api' is fully modifiable in 'Ep'
52 0.9 :: \+api_ac(Api) :- ep_no_ac(Ep), modifiable_context(Ep, Api, Id).
53 0.9 :: api_ac(Api) :- ep_ac(Ep), modifiable_context(Ep, Api, Id).
54
55 % Rule 10:
56 % 'non_modifiable_context' denotes the observation that the context of 'Api'is non-modifiable in 'Ep'
57 0.1 :: \+api_ac(Api) :- ep_no_ac(Ep), non_modifiable_context(Ep, Api, Id).
58 0.9 :: api_ac(Api) :- ep_ac(Ep), non_modifiable_context(Ep, Api, Id).
59
60 % Rule 11:
61 % 'user_modifiable_context' denotes the observation that the context of 'Api' is modifiable through user input to '
      Ep'
62 0.8 :: \+api_ac(Api) :- ep_no_ac(Ep), user_modifiable_context(Ep, Api, Id).
63 0.8 :: api_ac(Api) :- ep_ac(Ep), user_modifiable_context(Ep, Api, Id).
64
65 % Rule 12:
66 % 'special_input' denotes the observation that 'Ep' is taking a special input value as an argument that may be
      interpreted as access control (e.g., userid or appid).
67 0.1 :: api_ac(Api) :- ep_no_ac(Ep), special_input(Ep, Api, Id).
68 0.9 :: api_ac(Api) :- ep_ac(Ep), special_input(Ep, Api, Id).
```

Listing A.1: ProbLog Rules Listing

A.3 Samples of APIs from Synthetic Dataset

Here, we provide a sample of the APIs that implement strong access control and that were selected for the synthetic dataset construction, discussed in Subsection 5.5.3. The samples were chosen from Pixel 6 (V 11).

- FaceService.remove
- FaceService.setFeature
- ColorDisplayService.setSaturationLevel
- NetworkScoreService.requestScores

- SystemUpdateManagerService.retrieveSystemUpdateInfo
- StorageStatsService.queryStatsForUser
- StorageManagerService.decryptStorage
- StorageManagerService.encryptStorage
- MediaRouterService.getSystemSessionInfo

Appendix B

Ariadne

B.1 Similarity Between Two Nodes of the AC Dependency Graph

We define the the similarity score between two node n_1 and n_2 as follows:

$$\begin{aligned} sim\_score(n_1, n_2) = 0.25 \times type\_sim(n1, n2) \\ + 0.25 \times naming\_sim(n1, n2) \\ + 0.25 \times keyword\_sim(n1, n2) \\ + 0.25 \times method\_sim(n1, n2) \end{aligned}$$

The equation above uses four attributes for calculating similarity score, where each attribute is used as follows to derive a unique similarity value between 0 and 1:

$$type\_sim = \begin{cases} 1 & \text{if types are same} \\ 0.75 & \text{if Java collection types are same} \\ 0.5 & \text{if normalized types are same} \\ 0 & \text{otherwise} \end{cases}$$

$$naming\_sim = \text{Levenshtein distance between variable names of n1 and n2}$$

$$keyword\_sim = \begin{cases} 1 & \text{if same modifier and keywords} \\ 0.5 & \text{if same modifier or keywords} \\ 0 & \text{otherwise} \end{cases}$$

$$method\_sim = \text{Normalized Levenshtein distance between names of methods operating on n1 and n2}$$

B.2 Additional Case Studies

Probing recently running apps. Vivo defines a custom API within the AOSP service `ActivityTaskManagerService` called `getLastResumedActivity` that allows reading a member `packageName`, defined within a complex data holder `ActivityRecord`. Ariadne concluded that this field should be protected, as it identified `ActivityRecord`'s other members reachable via protected APIs. They share similar names with the target, resulting in a high confidence recommendation. The case allows reading sensitive information related to other packages;traditionally, AOSP requires a permission to access similar data. We have reported it to Vivo.

B.3 Edge Types

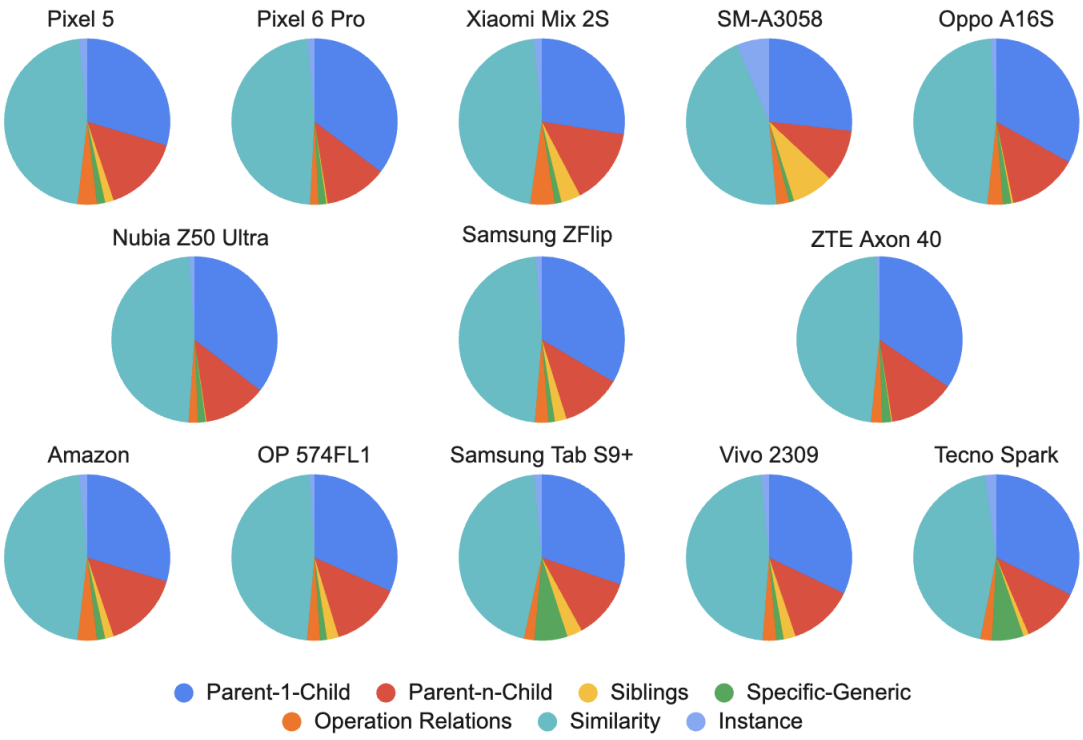

Figure B.1: Distribution of Edge Labels

Appendix C

RepFinder

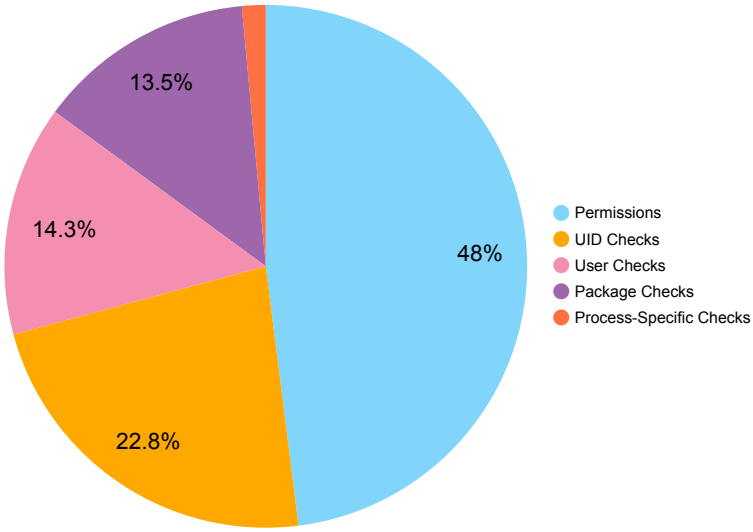

Figure C.1: Distribution of Access Control Checks

Distribution of Access Control Inconsistencies. RepFinder uncovers inconsistencies caused by various misuses of security features. A breakdown is depicted in Figure C.1. 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.