Systematically Detecting Access Control Flaws in the Android Framework

by

Zeinab El-Rewini

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

Waterloo, Ontario, Canada, 2022

Author's Declaration

Statement of Contributions

This thesis is based on two papers that I co-authored. Chapter 4 and portions of Chapter 3 are taken from [44], which was co-authored by my supervisor, Professor Yousra Aafer, and accepted for publication in ACM CCS 2021 [26].

Chapter 5 and portions of Chapter 3 are taken from a submitted paper that is co-authored by Professor Yousra Aafer and Zhuo Zhang, a Ph.D. student at Purdue University.

Abstract

Android's permission model is used to regulate access to the Application Program Interfaces (APIs) within the Android system services, which provide access to sensitive system resources, such as the camera and microphone. To successfully invoke sensitive APIs, a caller must hold one or more Android permissions.

Like all access control systems, the Android permission model is vulnerable to anomalies in security policy enforcement, including inconsistent access control enforcement. These inconsistencies occur when there are multiple paths to a sensitive resource, some with stronger access control enforcement than others. Attackers can exploit an inconsistency to improperly access a sensitive resource by taking the path with the weakest access control checks.

Many access control anomalies are a natural byproduct of the fragmented Android ecosystem, in which various vendors and carriers customize the baseline Android Open Source Project (AOSP) code base for their unique business needs. One consequence of this customization is software bloat, which is known to expand the attack surface. Though the security impacts of customization in the Android ecosystem have been studied extensively, the literature is missing a study on customization-induced code bloat and its effect on Android access control flaws.

Additionally, though a significant body of research has been dedicated to Android access control inconsistency detection, the existing state-of-the-art tools experience high false positive rates, as they precisely link access control checks to resources. That is, if is a sensitive resource is shown to be control-dependent on an access control check, the existing tools consider that check required for that resource with full confidence. In practice, this assumption is faulty as an access control check may not target all control-dependent resources.

In this thesis, we make two significant contributions to address both gaps in the literature. First, we conduct the first large-scale longitudinal study analyzing the security impact of Residual APIs, which are unused custom APIs that have been forgotten over the course of a customized AOSP code base's evolution. We find that Residuals are prevalent in the code bases of all major Original Equipment Manufacturers (OEMs) and that they result in security-critical vulnerabilities, including cases of inconsistent access control enforcement.

Second, we introduce a novel probabilistic inconsistency detection approach that introduces a measure of uncertainty to the linkage between resources and access control checks. Our approach uncovers implicit relations between framework-level resources and protections and leverages probabilistic inference techniques to generate recommendations that link resources to protections with a degree of uncertainty. We find that our approach improves existing tools by reducing false positives.

Acknowledgements

I would like to thank my supervisor, Professor Yousra Aafer, for her support and guidance throughout my master's studies. Thank you also to my thesis readers, Professors N. Asokan and Urs Hengartner, for their invaluable feedback.

I greatly appreciate Parjanya Vyas's helpful presence in the early days of our static analysis exploration. Additionally, I am thankful for Zhuo Zhang's generosity in sharing his probabilistic inference knowledge.

In particular, I would like to thank my parents, Hesham El-Rewini and Sherine Talaat, and my siblings, Bassel and Yassmine, for their love and patience.

Dedication

To Baba, Mama, Bassel, and Muni, my best friends.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the latter 2000s, the mobile operating system (OS) scene witnessed the emergence of Android, a new open-source mobile operating system touted as the first "truly open and comprehensive platform for mobile devices" [28]. Android was introduced by the Open Handset Alliance, a consortium of technology firms that imagined the OS as an innovative and cost-effective alternative to existing proprietary operating systems [27]. Despite an inauspicious start with the release of the HTC Dream/T-Mobile G1, which received mixed reviews [38], Android quickly overtook iOS to become the most popular mobile operating system in the world. As of May 2022, it held a worldwide market share of 71.45% for mobile operating systems [74].

Unfortunately, Android's rise in popularity has made it an attractive target for threat actors. In recent years, a number of very serious vulnerabilities in the Android OS have been discovered. Several Android system service APIs were found to be vulnerable to privilege escalation and information disclosure attacks [17, 14, 16, 15]. The Stagefright vulnerabilities [43] allowed attackers to remotely execute malicious code on Android devices. Similarly, the BlueFrag [68] vulnerability enabled zero-click remote code execution attacks to be conducted using Bluetooth. More recently, security researchers have discovered Android's susceptibility to spyware installed at the direction of nation-state actors. This includes the NSO Group's Pegasus spyware for Android (known as Chrysaor) [63] as well as Cytrox's Predator spyware [61].

To address these grave security challenges, researchers have been working to systematically detect vulnerabilities at all levels of the Android software stack, from the top-most application layer [55, 85, 59] to the underlying Linux kernel [72, 39, 66]. In particular, the Android framework layer has received significant research attention [29, 46, 31, 22, 41, 70, 21, 50] since this layer relies on the Android permission model to regulate access to sensitive system resources, such as the microphone and camera.

Researchers have also been working to understand how Android security vulnerabilities arise. Several studies [94, 25, 56] have demonstrated that the extensive customization of the Android Open Source Project (AOSP) plays a major role. Before making its way into consumer devices, AOSP is passed through a lengthy customization pipeline. Hardware manufacturers, device manufacturers and mobile carriers all customize the code base according to their company-specific priorities [24]. This customization has resulted in an extremely fragmented Android ecosystem that is conducive to the proliferation of security flaws. One class of such flaws is the hanging attribute reference[24], which is a reference to an undefined attribute, such as a package name. Attackers can re-define such an attribute to bypass security checks. Other customization-induced security flaws include vulnerable pre-loaded applications [81] and patch delay [54].

One of the consequences of customization is code bloat, which has both performance and security implications. Although the security impacts of Android customization have been extensively studied, no prior studies have examined the effects of customization-induced software bloat in the context of Android security.

To address this gap, we investigate the security impact of API bloat in custom Android ROMs. We develop ReM, a suite of static analysis techniques to detect custom private APIs that are defined but not used. We refer to these bloated APIs as *Residuals*. We perform the first large-scale, cross-version study of Residual APIs and find that they have a prominent presence in customized AOSP, comprising up to 42% of private APIs in some models. We also find that 23% of Residuals lead to serious access control flaws including deprecated security attributes, which attackers can re-define to their advantage, and access control inconsistencies that expose sensitive resources.

One such flaw is inconsistent security policy enforcement, which occurs when one path to an API has stronger access control requirements than another. These flaws are common in customized AOSP, as access control enforced within vanilla AOSP may be watered down or completely missing after customization. Prior work [21, 70] has shown that inconsistent permission enforcement allows attackers to perform a range of attacks, including denial-of-service, privilege escalation and battery drainage attacks.

To reduce the prevalence of access control inconsistencies, researchers have proposed a number of techniques that involve systematically analyzing both vanilla and custom AOSP. These include permission mapping approaches [29, 46, 31, 22, 41], which deduce the required access control enforcement for framework resources, and access control inconsistency detection approaches [70, 21, 50], which identify cases of inconsistent access control enforcement.

In our analysis of inconsistent access control enforcement within Residual APIs, we observed inherent limitations in the existing inconsistency detection approaches. First and foremost, the existing approaches consider an access control check p to be required for an API a if a is control-dependent on p. However, we discover that existing approaches neglect to consider other implicit relations beyond control dependence. Additionally, the existing tools treat the API-to-access control linkage as precise when, in reality, this linkage is always associated with some degree of uncertainty.

To address these limitations, we introduce a novel probabilistic approach that can be used to recommend access control enforcement for framework-level resources, which include APIs, field accesses and internal methods. This approach considers the linkage between a resource and an access control enforcement to be *inherently uncertain* and thus reduces the false positives seen in previous inconsistency detection works. Additionally, we consider a wide range of implicit relations that help probabilistically deduce that a resource and protection should be linked.

1.1 Our Contributions

We make the following contributions:

- We present ReM, a set of new analysis techniques that detect and evaluate the risks associated with Residuals, custom APIs that are no longer used but continue to be defined in a code base. Our techniques are specially tailored to detect evolution-induced access control vulnerabilities. Using ReM, we conduct the first systematic large-scale study of Residuals. Our study unveils the extent and prevalence of Residuals and, more importantly, demonstrates that Residuals do indeed open the door to various attack vectors. We were able to exploit eight different Residuals to develop keyloggers, perform data injection attacks and even launch activities with system privilege.

- We introduce Poirot, a tool that generates probabilistic protection recommendations for Android framework resources. Poirot relies on both probabilistic inference and static program analysis to account for the uncertainties pertaining to access control implementations. The tool supplements the traditional reachability analysis with seven semantic, structural and data-flow relations that provide insight into the relationships between framework resources and protections. We discuss our evaluation of Poirot, which sheds light on the advantages of a probabilistic approach in reducing false positives. We also present a number of Poirot-identified inconsistencies that we were able to exploit, including one inconsistency that allowed us to crash and reboot the system.

1.2 Organization

In Chapter 2, we provide a general background on the Android ecosystem, software stack and access control mechanisms. In Chapter 3, we discuss related work on vendor customization and the detection of access control flaws in Android. In Chapter 4, we move to present ReM and our large-scale study of bloated Residual APIs in custom Android ROMs. Chapter 5 discusses our next-generation inconsistency detection tool, Poirot. Finally, we provide concluding remarks in Chapter 6.

Chapter 2

Background

In this chapter, we provide a general background on Android. We begin by discussing the Android ecosystem in Section 2.1 before moving to an overview of the Android software stack in Section 2.2. Finally, Section 2.3 discusses Android access control mechanisms and presents an example of inconsistent access control enforcement.

2.1 Ecosystem

Before reaching consumers, the Android operating system undergoes an extensive customization process. A typical upgrade operation includes customization at multiple points along the supply chain. In [58], Jones et al. describe this process in detail for security updates and operating system updates. The general pattern is that changes are first made by Google and chipset vendors. Then Original Equipment Manufacturers (OEMs), such as Samsung or Lenovo, make customizations in conjunction with mobile carriers, such as Bell or T-Mobile. Both OEMs and carriers are responsible for providing the end product to consumers. Figure 2.1 illustrates the diverse Android ecosystem and indicates that even a single OEM may implement different customizations based on the device model and version, adding further fragmentation.

To address the fragmentation problem, Google has deployed a number of initiatives, including Android One, Project Treble and Project Mainline [58]. Android One is a program where certain device models run a barely-customized version of Android in exchange for some years worth of OS upgrades and security updates. Project Treble separates vendor implementations from the stock AOSP, while Project Mainline allows end users to

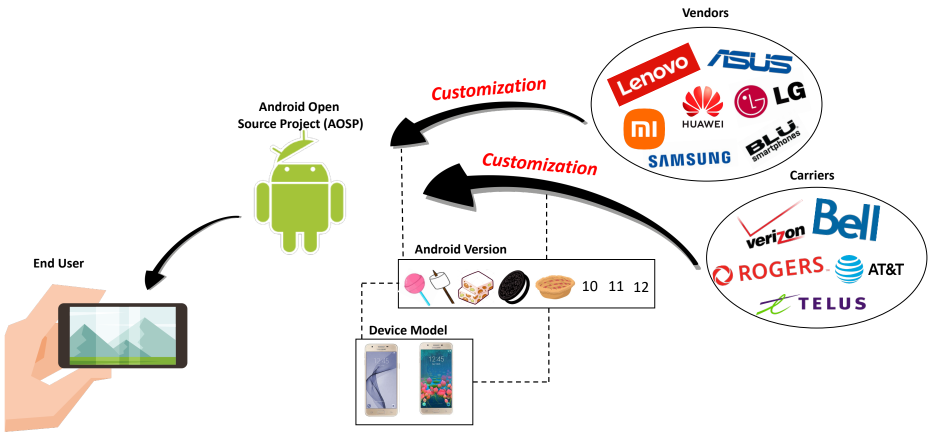


Figure 2.1: The Android Ecosystem

update system components through Google Play without going through the OEMs. Despite these initiatives, AOSP customization is still under-regulated and regularly results in access control flaws.

2.2 Software Stack

Figure 2.2 presents an overview of the Android software stack. The top-most layer is the Android application layer, which consists of both system and third-party applications [2]. Below the application layer is the Android middleware, which contains native C/C++ libraries, the Android Runtime and the Android framework [31]. The Android framework contains a collection of Java-based libraries and services that implement the Android application program interfaces (APIs) [29]. Each service within the framework allows the application access to a specific system resource [31].

Binder, Android's primary inter-process communication mechanism, can be used to communicate with the framework services [31]. An interface must first be defined in Android Interface Definition Language (AIDL). An AIDL compiler can then be used to generate `Proxy` and `Stub` classes. During a remote procedure call, a `Proxy`, which is located at the client-side, can transform the parameters into primitive objects that can then be marshalled across process boundaries [32]. A `Stub` can then unmarshall the data and call

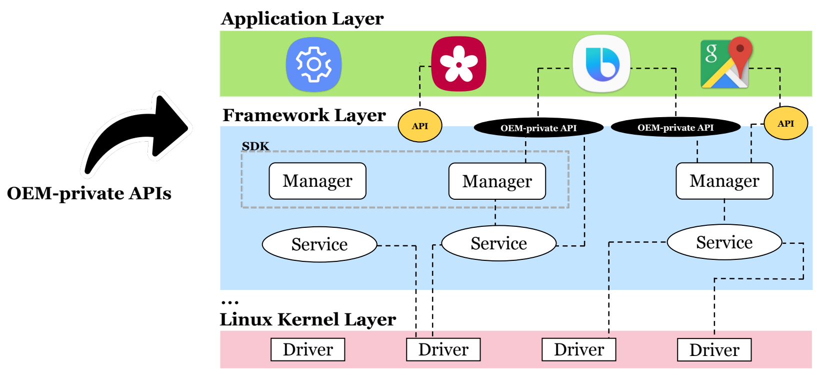


Figure 2.2: The Android Software Stack

the target method.

Instead of using a `Proxy` to communicate with a system service, a developer could use a `Manager` provided within the Android Software Development Kit (SDK). Each `Manager` simply wraps around a `Proxy` [31]. Ultimately, developers do not have to use either the `Manager` or the `Proxy` classes. Instead, they can access system services directly using Binder transaction IDs. For more information on communication through transaction Ids, see Section 4.1 in [23].

2.3 Access Control Mechanisms

The Android system services contain sensitive APIs meant to be invoked only by privileged callers. To regulate access to these APIs, Android relies on a high-level permission model. Calling sensitive framework service APIs requires the caller to hold a specific set of permissions, which are strings tied to a caller's UID. In Android, most permissions are categorized as either granted at install-time or run-time [1]. Install-time permissions can be further categorized as either normal permissions or signature permissions [77]. Normal permissions protect access to low-sensitivity operations and data, while signature permissions allow access to an application's private data. Only applications signed with a matching certificate to the application that defined the signature permission would be granted access. Run-time permissions consist of dangerous permissions, which protect sensitive data and resources, such as the camera and microphone [1].

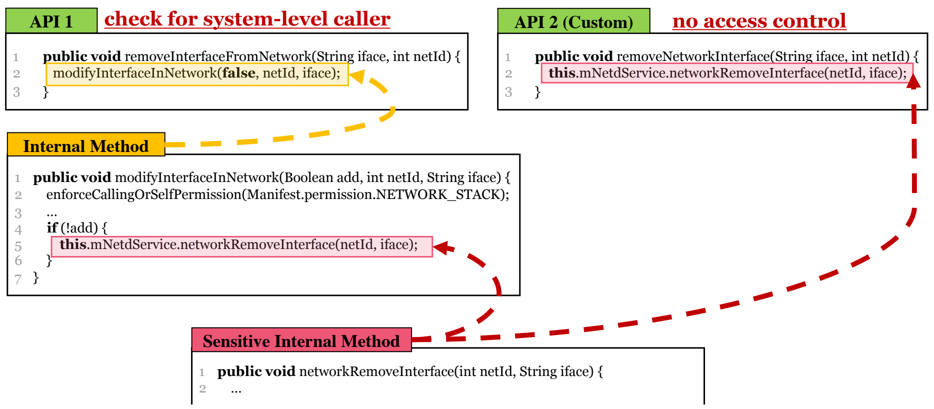
To check whether a caller holds the proper permission(s), a call to a variation of `checkPermission()` may be used. Alternatively, the caller's UID can be checked using

```

private void enforceAccessRestrictions() {
        int uid = Binder.getCallingUid();
        if (uid == Process.SYSTEM_UID || uid == Process.myUid() || uid == Process.PHONE_UID) {
            return;
        }
        String defaultDialerPackageName = getContext().getSystemService(TelecomManager.class)
                .getDefaultDialerPackage();
        if (TextUtils.isEmpty(defaultDialerPackageName)) {
            throw new SecurityException("Access to call composer locations is only allowed for the"
                    + " default dialer, but the default dialer is unset");
        }
```

Figure 2.3: Checking a Caller's Uid

`getCallingUid()`. For instance, Figure 2.3, which is taken from AOSP's `CallComposerLocationProvider` class, presents a method that returns without issue if the caller's UID matches a system UID, the UID of the current process or the UID used by telephony processes. Otherwise, it throws a `SecurityException`. Similarly, the calling PID (process Id) can also be checked using `getCallingPid()`. Since Android allows the creation of multiple user profiles, another commonly used restrictive check is `getUserId()`, which returns a value indicating which profile the call originated from. AppOps permissions allow more fine-grained control of operations on a per-app basis. Users can mark certain operations as "granted" or "restricted" and the `AppOpsService` determines at run-time whether a given operation is permitted [33].



```
public void removeInterfaceFromNetwork(String iface, int netId) {
    modifyInterfaceInNetwork(false, netId, iface);
}
```

```
public void removeNetworkInterface(String iface, int netId) {
    this.mNetdService.networkRemoveInterface(netId, iface);
}
```

```
public void modifyInterfaceInNetwork(Boolean add, int netId, String iface) {
    enforceCallingOrSelfPermission(Manifest.permission.NETWORK_STACK);
    ...
    if (!add) {
        this.mNetdService.networkRemoveInterface(netId, iface);
    }
}
```

```
public void networkRemoveInterface(int netId, String iface) {
    ...
```

Figure 2.4: Demonstrating an Inconsistent Access Control Enforcement

The access control enforcement mechanisms described thus far are all used at or above the framework layer. However, Android also includes lower-level access control [53]. The kernel layer incorporates Discretionary Access Control (DAC), Mandatory Access Control

(MAC) and Linux capabilities. Android's DAC is realized through the use of UIDs and GIDs, with dynamically installed third-party applications each receiving a unique UID. Like Linux, Android allows highly privileged processes to carry capabilities with a subset of the root user's privileges. The MAC policy is specified through the use of SEAndroid, which is an Android extension of SELinux.

Now that we have discussed the various access control mechanisms leveraged by Android, we provide a simple example to demonstrate what exactly we mean by an access control inconsistency. Figure 2.4 depicts two paths taken from two APIs to a sensitive internal method, `mNetdService.networkRemoveInterface()`. The path starting from `NetworkManagementService.removeInterfaceFromNetwork()` requires the caller to hold the system-level permission `CONNECTIVITY_INTERNAL`. On the other hand, the path starting from the fictional `CustomNetworkManagementService.removeNetworkInterface()` does not require any permission enforcement. Consequently, if attackers wish to invoke the convergence point `mNetdService.networkRemoveInterface()`, they can simply invoke the lesser protected API within the `CustomNetworkManagementService`. We note that, in practice, the weaker path may enforce a subset of the access control required by the most highly-protected path.

Chapter 3

Related Work

We split the Related Work chapter into two sections. In Section 3.1, we discuss research on the impact of vendor customization. In Section 3.2, we present details of diverse approaches to the detection of access control flaws.

3.1 Vendor Customization

Vendors extensively customize device drivers, system applications and system services [56]. Since this customization is unregulated, it often introduces new security risks. A number of works have examined the security impacts of this customization. At the top-level layers of the Android software stack, Gallo et al. [48] analyze five custom Android distributions and identify that customization results in an expanded attack surface and poor permission usage. They go as far as to suggest that security-conscious consumers should avoid heavily-customized Android distributions. In [24], Aafer et al. detect hanging attribute references, which can occur when customization results in references to nonexistent attributes that can then be defined by a malicious party. They discover tens of thousands of these hanging attribute references in a study of almost 100 ROMs. Zhang et al. introduce InVetter [89], a tool that identifies weakened input validation checks within customized system services. They are able to identify twenty serious system service vulnerabilities. Iannillo et al. test custom framework services through their tool, Chizpurfle [40], a greybox fuzzer used to uncover vulnerabilities in customized framework services.

Other works target the deeper layers of the Android software stack. Possemato et al. [67] analyze thousands of Android ROMs to determine the effects of customization on compliance with Google's guidelines and general Android security. After their analysis of the ROMs' binary customization, init policies, SELinux scripts and kernel security, they conclude that existing efforts by Google to reduce fragmentation do not go far enough. At the Linux-layer, Zhou et al. [94] evaluate problematic vendor modifications to Linux device drivers. They rely on dynamic analysis to identify which Linux files are accessed by device operations. They then investigate whether customized Linux device drivers result in under-protected files and find many flaws related to the driver customization. Hay [52] analyzes the security of customized Android bootloaders. Specifically, Hay examines the fastboot interface implemented by the Android Applications Bootloader (ABOOT). Hay finds that, in customized bootloaders, it is possible to tamper with the archive containing the `init` process, which is the first user space process. Zhang et al. [88] explore the impact of customization on the ION unified memory management interface used in ARM-based Android devices. Using a combination of static and dynamic analysis, they find that the customization of ION can open the door to memory dumping and denial-of-service attacks. Yu et al. [87] examine the effects of customization on SEAndroid policies and find that unregulated, customized SEAndroid policies are prevalent. They propose SEPAL, a tool that relies on Natural Language Processing (NLP) techniques to determine whether a custom policy is unregulated.

Though software bloat is one consequence of the fragmented Android ecosystem, work on software debloating from a security angle is noticeably limited in the Android context. One previous tool, RedDroid [57], debloats Android applications. However, the corresponding study does not delve deeply into the unique access control flaws caused by bloatware in Android. To our knowledge, no prior Android-specific studies specifically examine the access control impact of customization-induced software bloat. In fact, the majority of works exploring the security benefits of software debloating are focused on web applications. Azad et al. [30] explore the server-side, using dynamic profiling to identify code that should not be removed during the debloating process. On the other hand, Schwarz et al. [69] and Snyder et al. [71] focus on client-side browser security. Others, such as Mururu et al. [65] present binary debloating approaches, while Brown and Pande [35] [36] propose a new tool and metrics to assess the security impact of debloating. Our work on ReM is the first to examine the impact of Residual APIs on Android security.

3.2 Access Control Flaw Detection

Permission mapping and inconsistency detection are two major areas of interest to researchers concerned with detecting access control flaws. Permission mapping tools attempt to deduce the set of required permissions for framework APIs to prevent permission under-privilege as well as permission over-privilege. On the other hand, inconsistency-detection tools identify resources with inconsistent access control enforcement. In this section, we begin by discussing the existing work in both areas. We then move to a discussion on probabilistic inference techniques, which can be used to improve the state-of-the art tools used to detect access control flaws.

Permission maps compensate for Android's non-existent permission specification. They provide linkages between an API and that API's required access control enforcement. One of the earliest permission mapping tools is Felt et al.'s Stowaway [46], which uses a dynamic, feedback-directed testing approach to determine the maximum set of permissions that an Android application requires. To build upon the efforts of Stowaway, Au et al. [29] develop Permission Scout (PScout), a version-independent permission mapping tool that relies on static analysis techniques. PScout uses a reachability analysis and is thus able to cover more of the Android framework than Stowaway. Like Stowaway, the tool is conservative, assuming that a union of all possible permission sets is required. Backes et al.'s Axplorer [31] creates a static model of the Android framework that attempts to approximate the behavior of the threading mechanisms relied upon by framework services.

Axplorer, Stowaway, and PScout all over-approximate the number of permissions required for a given API. Aafer et al.'s Arcade [22] tool addresses these limitations by generating a path-sensitive permission map that can be used to deduce an API's minimum required permissions. Dynamo [41], the latest permission mapping tool, relies on a grey-box fuzzing technique and run-time instrumentation to detect permission checks and gain coverage information.

In addition to associating resources with protections, significant research effort has been dedicated to pinpointing security policy inconsistencies in the Android framework. Shao et al.'s Kratos [70] is among the first Android-specific security policy inconsistency detection tools. Kratos over-approximates access-control inconsistencies, as it relies on a path-insensitive analysis and handles only four specific types of security checks. AceDroid [21] models a more diverse array of security checks and incorporates a novel normalization mechanism to avoid detecting inconsistencies arising from syntax differences. The Authorization Check Miner (ACMiner) tool [50] provides a semi-automated approach that identifies security checks through the intuition that the existence of a path from a framework entry point to a `SecurityException` implies the existence of another path along which access to a protected resource is granted.

Two recent works extend their scope beyond the Android framework. FReD [18] identifies inconsistencies in API access control requirements by analyzing Linux-layer permissions, while IAceFinder [93] detects cross-context inconsistencies in the Java and Native layers.

While this body of literature has proven to be quite beneficial, we note that the existing works suffer from shortcomings. Cross-layer inconsistency-based solutions are limited in scope, as they can only detect vulnerabilities in APIs with specific implementations (i.e, APIs accessing files as in FReD [18] or APIs reaching a JNI interface as in IAceFinder [93]). Though in-framework inconsistency detection approaches leverage a richer learning ground for access control owing to the substantial amount of reachable resources in the framework-layer, we note that their underlying detection methodology is highly-simplistic, often leading to inaccurate output unless substantial heuristics are adopted. Specifically, the tools are founded on the assumption that two APIs converging on an instruction (i.e., field update, method invocation) are related and thus require similar protections. However, we note that the convergence point may be auxiliary to the general promised functionality and hence irrelevant to the enforced access control. Failing to discern the relevance of the convergence point leads to significant false positives.

Our tool, Poirot, is the first to re-conceptualize the inconsistency detection problem by using probabilistic inference to account for uncertainty. We are inspired by the wide applications of probabilistic inference techniques. Probabilistic type inference [83] has been proposed for dynamic programming languages such as Python. Probabilistic model checking [42, 60, 47] enhances the existing deterministic techniques by encoding probabilities into the transition among states. With largely extended scalability, probabilistic symbolic execution [49, 34] predicts the likelihood of reaching a certain program point. Researchers also adapt inference and distribution analysis techniques in binary analysis [90, 64] to provide a systematic approach to model the inherent uncertainty caused by information loss during compilation. Other applications include fuzzing [91], network trace analysis [86], race/leak detection [37, 51] and runtime event analysis for program understanding [92, 76].

Probabilistic inference has also been adopted for vulnerability detection and security invariant validation. Engler et al. [45] devise a static checker to infer bugs in real systems such as Linux and OpenBSD. AutoISES [75] automatically infers high-level security specification and detects violation afterwards. Srivastava et al. [73] adapt a precise, flow- and context-sensitive security policy inference technique to analyze relationships between security checks and security-sensitive events. Vaughan et al. [78] devise a security-expressive language to describe security policy where inference of expressive is introduced to help reduce the number of annotations. JIGSAW [79] infers programmer expectations to achieve better access control. Yamaguchi et al. [84] leverage inference techniques to search taint-style vulnerabilities in C code. Our tool, Poirot, adopts rule inference techniques to recommend Android access control.

Chapter 4

ReM

4.1 Introduction

Evolution-related vulnerabilities are introduced when OEMs cannot respect well-established security requirements while keeping up with the fast pace of Android version updates and the sophistication of new functional requirements. For each new Android version and device model, OEM developers adapt the existing custom codebases to the new requirements by adding or removing custom functionalities – eventually introducing new OEM-specific private APIs and removing unused ones. From a security standpoint, removing unused private APIs, which we name *Residuals*, is highly important. Unused functionality not only increases code complexity but also broadens the attack surface. Many serious software vulnerabilities in commodity software and platforms are rooted in features that are never used [3].

Several research efforts [62, 82] have been proposed to investigate the phenomena of unneeded API removal from Android codebases, including deprecation practices, developer reactions and compatibility aftermath. However, to the best of our knowledge, no effort has looked into the security implications of failing to remove them. In this section, we bridge the gap by performing a large-scale security investigation of Residual APIs. Our study aims to answer whether Android Residuals *do unnecessarily open the door to security flaws* as in other software and platforms.

To conduct the study, we put forward a solution that detects Residuals and evaluates their access control enforcement within custom ROMs. Our tool entails extensive program analysis of a large corpus of custom APIs (26,883), defined over our collection of 628 ROMs.

Intuitively, a Residual API can be defined as any private API that is not used on a particular device but is used in earlier versions and/or in other models. This definition oversimplifies the nature of Residuals in Android. A seemingly unused API may be indirectly called through complex call chains and reachable through multiple framework entry points. To ensure accurate Residual detection, our analysis attempts to recover framework entry points through a specialized backward search over the framework classes.

The above definition further implies that the mere occurrence of unused APIs in a few isolated, random ROMs – without accounting for the APIs' *historical* and *model-specific* use patterns – may not accurately signal a Residual's presence. Our approach addresses this issue by building a usage history of custom APIs over our curated ROM samples. Specifically, Historical Residuals are detected by observing gradually or abruptly retiring APIs over time, while Model Residuals are identified by looking for specific use within clusters of devices from the same model or series.

To understand the security risks a Residual may pose, we perform a thorough security analysis. Our proposed analysis focuses on evaluating access control enforcement adopted by Residuals. The evaluation is guided by the intuition that, through various releases, Android APIs naturally evolve to add, fix and modify existing access control to patch vulnerabilities or add additional security requirements. Any failure to keep up with access control evolution will inevitably introduce anomalies and potential vulnerabilities. On the one hand, failing to adapt to the unstable device-specific implications of Android security features (e.g., permissions) will inevitably introduce security flaws. On the other hand, failing to keep up with the ever-evolving Android access control mechanism will lead to the adoption of obsolete security enforcement – thus unnecessarily re-opening the door to older vulnerabilities and invalidating current security requirements. Our proposed solution evaluates Residuals by inspecting implemented access control enforcement and verifying that it adopts *sound* security features and reflects *up-to-date* security requirements.

4.2 Organization

In Section 4.3, we provide a brief background on Residual APIs and the tactics used to safeguard them. We then move to investigating the dangers posed by Residuals in Section 4.4. Sections 4.5 - 4.7 introduce our approach to Residual detection. In Section 4.8, we evaluate the security properties of identified Residuals. Subsequently, Section 4.9 presents the results of our large-scale measurement study of Residuals in the Android ecosystem. We move to a discussion on the security landscape of Android Residuals in Section 4.10. Finally, in Section 4.11 we discuss our exploitation of vulnerabilities in Residual APIs.

4.3 Residual APIs

OEMs aggressively customize the AOSP baselines. For each new Android version and device model, OEM developers adapt their codebases to new functional requirements by adding, altering and removing APIs. This extensive API retrofitting process usually spans Android SDK APIs as well as OEM-specific private APIs. When retrofitting SDK APIs, OEMs must abide by Google's regulations to meet compatibility requirements. That is, APIs designated for use, deprecation and removal by Google should be similarly designated by OEMs.

However, when it comes to OEM private APIs, the process is less regulated. Private APIs, provided to support internal framework and preloaded app developers, are added and removed frequently (∼880 and 92 times, respectively, as reported in our dataset). This under-regulation coupled with OEMs' efforts to provide a one-size-fits-all framework implementation contributes to the production of *bloated* custom codebases. OEM devices tend to include a substantial number of private APIs [70, 21] (reaching up to ∼3,500 in Samsung versions 7.0.1 and 8.1), some of which do not even fit with the devices' functional requirements. We refer to such unused APIs as *Residuals*.

Residuals not only increase code complexity but also induce compatibility issues. For instance, invoking an unsupported API on a particular device will lead to app or system crashes. Even worse, when Residuals provide sensitive operations and are not properly protected, they *unnecessarily* induce security issues. This is particularly inevitable when OEMs fail to adapt *up-to-date* and *compatible* security checks to safeguard a Residual's functionality.

To deal with compatibility issues and to properly protect a Residual's functionality, OEM developers implement safeguards. At a high level, the guards attempt to reduce the pool of devices on which a Residual may be activated or restrict the callers to a set of verified entities (the expected users of the Residual). Specifically, the safeguards fall into the following two categories:

(1) Configuration Checks: These guards are adopted to ensure that an API's provided functionality is compatible with the current release and/or is supported on the running platform. For example, for legacy APIs targeting obsolete functionalities, the guards make sure that the functionality cannot be triggered in newer releases. Similarly, for APIs supporting specific capabilities, the configuration guards ensure that the running platform embeds the corresponding hardware. Figure 4.1 depicts a few configuration checks implemented by LG within a custom API assisting AT&T tethering. Lines 2-3 ensure that the API can only be triggered on devices with mobile data capabilities; that is, if the device

```
1  boolean startATTEntitleforTethering(...){
2      if(SystemProperties.get("ro.build.characteristics").equals("tablet")){
3          if(!telephonyManager.hasIccCard()){
4              Log.d("WifiService", "tablet has no sim card");
5              return false;
6          }
7      if(SystemProperties.get("ro.build.target_country").equals("US")
8        && SystemProperties.get("ro.build.target_operator").equals("ATT"))
9              if(getAppName(Binder.getCallingUid()).equals("com.smartcom"))
10                 // perform actual functionality
```

Figure 4.1: Configuration Checks in a Custom LG API

is a tablet, it should embed a SIM card. Other checks at lines 7 and 8 verify that the functionality can only be triggered in devices operated by the US-based carrier AT&T. Even if the API is introduced on devices not conforming to these checks, its functionality is safeguarded.

(2) Access-Control Checks: These checks reflect traditional Android access control enforcement. In this scenario, they restrict access based on unforgeable properties (e.g., UID) or acquired permissions. The calling entities reflect system processes or preloaded apps that exist on the devices where the Residuals are active.

While certain access control checks are intrinsically sufficient to properly protect a Residual, we observe that other checks implicitly rely on a co-located configuration check for validity. Without this secondary configuration check, the access control may be totally flawed. Consider the check performed at line 9 of Figure 4.1, which verifies that the calling app matches the name *"com.smartcom."* Observe that this check is *unsound* by itself since a package name can be squatted. Unless the package exists on the device, any third-party app can claim to be *"com.smartcom"* and trigger the privileged functionality. In this case, we found that *"com.smartcom"* comes preloaded on AT&T models, implying that the package check is actually sufficient under AT&T builds. Hence, the configuration check at line 8 validates the soundness of the package check.

Given these intrinsic complex properties, coupled with the prevalence of Residuals and the fast-paced Android updates, we argue that ensuring proper and valid safeguards is challenging and error prone. Access control vulnerabilities may be *unnecessarily* introduced because of Residuals.

4.4 Problem

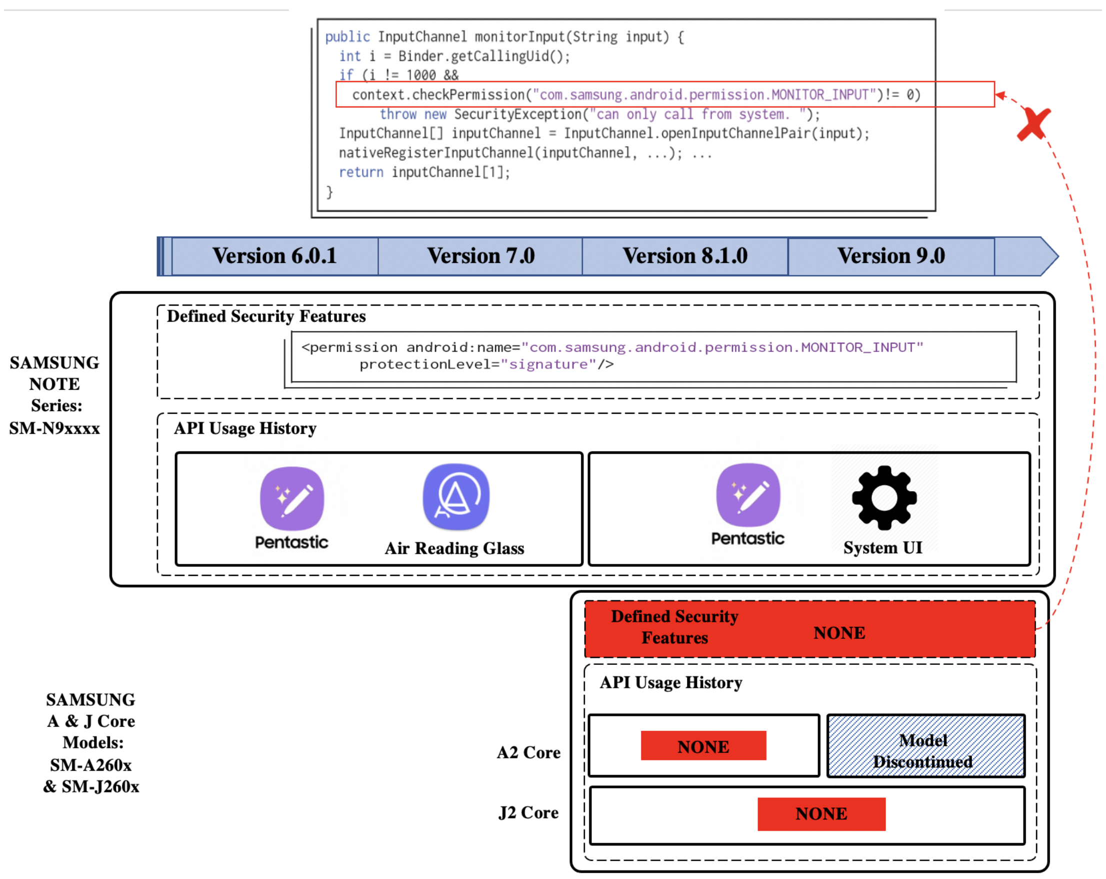
Since Residuals are deemed unnecessary and, at times, not intended for deployment on a particular device, framework developers may naturally overlook their implementation during integration and version upgrades. *Evolution-induced access control errors* are particularly dangerous in Residuals. On the one hand, a failure to account for the unstable device-specific implications of security features will inevitably introduce security flaws. On the other hand, a failure to keep up with the ever-evolving Android access control mechanism will lead to the adoption of obsolete security enforcement – thus unnecessarily re-opening the door to older vulnerabilities and invalidating current security requirements. Our study reveals a plethora of vulnerabilities resulting from these failures, including enabling third-party apps to exploit a Residual to access sensitive resources (such as the input driver).

4.4.1 Unsound Security Features

The correctness of access control enforcement heavily relies on the soundness of adopted security features (e.g., permission, calling uid, package name). While some security features are persistently sound (e.g., relying on the calling UID to verify that the caller is SYSTEM), others may imply different protections depending on the running device and model. Hence, if OEM developers do not account for these changes, a Residual API may use *incompatible* and *unsound* features that imply protections only available in other devices where the Residual is active.

Motivating Example

Consider the case depicted in Figure 4.2. As listed at the top, Samsung introduces a custom API `InputManager.monitorInput(...)` in a few device models. The API creates an input channel that receives input events from the input dispatcher. It can thus be used to intercept and monitor input events such as screen tap coordinates and key presses. Given the sensitivity of the operation, Samsung enforces high-privilege requirements. The caller must belong to the system process (enforced through the check `getCallingUID() = 1000`) or hold Samsung's custom permission `com.samsung.android.permission.MONITOR_INPUT`. Thus, a third-party app cannot invoke this API unless it can somehow obtain the permission.



```
public InputChannel monitorInput(String input) {
  int i = Binder.getCallingUid();
  if (i != 1000 &&
    context.checkPermission("com.samsung.android.permission.MONITOR_INPUT")!= 0)
      throw new SecurityException("can only call from system. ");
  InputChannel[] inputChannel = InputChannel.openInputChannelPair(input);
  nativeRegisterInputChannel(inputChannel, ...); ...
  return inputChannel[1];
}
```

```
<permission android:name="com.samsung.android.permission.MONITOR_INPUT"
      protectionLevel="signature"/>
```

Figure 4.2: Usage of Undefined Permissions in Residual APIs

The lower parts of Figure 4.2 depict the API's related security definitions and usage history in Samsung Note Series and A/J Core Series. As illustrated, in the Note Series, the API is used in versions 6.0.1 to 7.0 (from Oct'15 to Oct'16) by two preloaded apps: *Pentastic* and *Air Reading Glass*. Starting from versions 8.1 to 9 (from Dec'17 to Sep'19), the API is used by *Pentastic* and another preloaded app *System UI*. Observe that the devices define the API's required permission MONITOR_INPUT and designate it a *signature* level protection, which cannot be acquired by third-party apps.

In contrast, consider the usage history of the API in the Samsung A/J Core Series, illustrated at the bottom. As shown, the API is introduced in the first release of the devices (version 8.1, Dec'17) and has been consistently defined up to version 9 (Aug'18) in J2 Core (note that A2 Core was discontinued). However, no active usage site has ever been identified throughout the versions – thus making the API a Residual in A/J models. Though the API seemingly enforces access control checks, it is actually vulnerable. Since the API is deemed nonfunctional, the framework developers have overlooked defining the required permission (i.e., `MONITOR_INPUT`).

Since the permission is undefined, any entity that defines it can acquire it and subsequently trigger the Residual's privileged operation. In this particular case, we were able to exploit the Residual to develop a keylogger without any permission requirements. Observe that the vulnerability has been dormant since its introduction in Dec. 2017, in part because the API *has never been used* since then. The issue has been acknowledged and fixed by Samsung. We note that undefined security features may also occur in non-Residual APIs. However, as uncovered by our study, they are substantially more prevalent in Residuals (refer to Section 4.10.3).

4.4.2 Obsolete Access Control Enforcement

Android has expanded beyond the traditional smartphone to support other device types and use scenarios. Along with the expansion, new security features and requirements are incrementally added with each update. A failure to keep Residuals up-to-date and compliant with the new requirements can cause anomalies.

Motivating Example

Figure 4.3 depicts the access control evolution of two APIs: AOSP's `getDeviceId()` and LG's `getDeviceIdForVZW()`, both allowing the caller to read the device's ID (e.g., IMEI). We note that LG's API is defined in different models (versions 5-8), but is only used in VZW-specific models. Thus, it is a Residual in all other models.

As shown, AOSP's access control has evolved from enforcing a single *dangerous* permission `READ_PHONE_STATE` in versions 5.0-5.1.1 to requiring two different protections in versions 6.0 to 8.1 – either the permission `READ_PRIVILEGED_PHONE_STATE` or the permission `READ_PHONE_STATE` as well as explicit user approval indicated by an `AppOps` operation check. In contrast, LG's Residual has not seen a similar update, instead still adopting the obsolete single permission requirement. Under this anomaly, a malicious app could exploit the weakly protected Residual to read the device's IMEI. We have confirmed the vulnerability and reported it to LG. [1]

[1]The issue has been acknowledged by LG.

It is worth mentioning that starting from Android 10, Google prohibits third-party apps from accessing non-resettable identifiers such as the IMEI. AOSP's `getDeviceId` returns `NULL` in devices running 28 and older. Yet LG's Residual still returns a valid id.

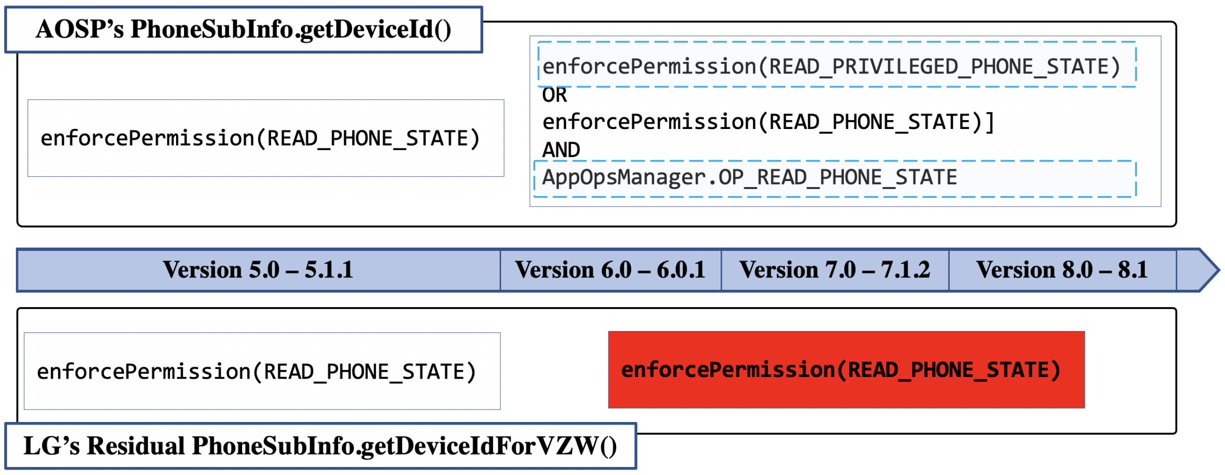

AOSP's PhoneSubInfo.getDeviceId()
enforcePermission(READ_PHONE_STATE)
enforcePermission(READ_PRIVILEGED_PHONE_STATE)
OR
enforcePermission(READ_PHONE_STATE)]
AND
AppOpsManager.OP_READ_PHONE_STATE
Version 5.0 – 5.1.1
Version 6.0 – 6.0.1
Version 7.0 – 7.1.2
Version 8.0 – 8.1
enforcePermission(READ_PHONE_STATE)
enforcePermission(READ_PHONE_STATE)
LG's Residual PhoneSubInfo.getDeviceIdForVZW()

Figure 4.3: Access Control Evolution of Two LG APIs

4.5 Our Solution

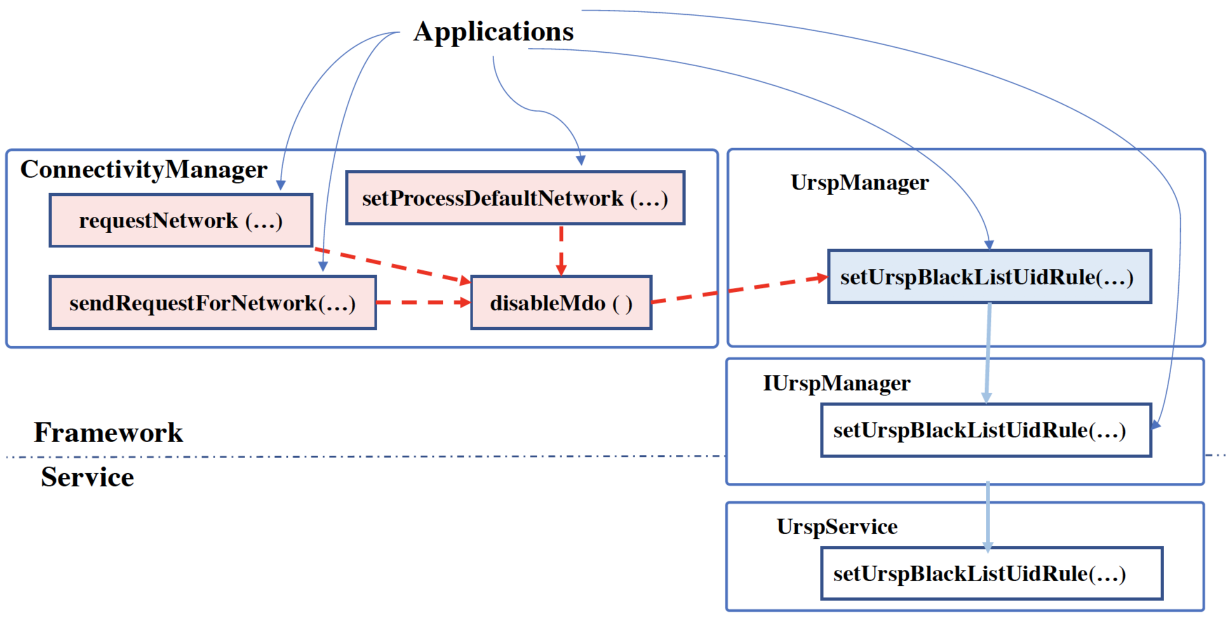
Our proposed investigation proceeds as follows. First, through program analysis of framework and preloaded apps, we recognize and pinpoint potential Residual instances in a ROM. We then build and investigate their usage patterns over a set of curated ROM samples. Confirmed Residuals (e.g., those following declining, retiring usage trends) are then fed to our proposed security analysis. We statically analyze the confirmed instances to identify the presence of unsound security features and obsolete access control checks.

The overall accuracy of the system relies heavily on the correct identification of Residual APIs. In light of custom call chains and API use patterns, detecting APIs that are defined but not used is not straightforward. Specifically, the detection entails the following two challenges:

Challenge 1: Identifying Entry Points Leading to a Target API

We identify through our analysis that custom APIs are often not directly invoked by other preloaded apps and framework services (hereafter referred to as components). Rather, they are usually wrapped in *Manager* APIs that are transitively wrapped around framework methods from both OEMs and Android; hence forming a long call chain from the components to custom APIs.

Consider the Samsung custom API `IUrspManager.setUrspBlackListUidRule(...)` shown in Figure 4.4.



Applications
ConnectivityManager
requestNetwork (...)
setProcessDefaultNetwork (...)
sendRequestForNetwork(...)
disableMdo ( )
UrspManager
setUrspBlackListUidRule(...)
IUrspManager
setUrspBlackListUidRule(...)
Framework
Service
UrspService
setUrspBlackListUidRule(...)

Figure 4.4: Multiple framework entry points leading to the custom API `IUrspManager.setUrspBlackListUidRule(...)`

The API is introduced in most SM-G38xxx models. As shown, it is wrapped in a custom Manager API `UrspManager.setUrspBlackListUidRule()`, which is transitively called by four other methods. The call chain is depicted by the dashed arrows. First, it is invoked directly by `disableMdo()`, a custom method added by Samsung to AOSP's `ConnectivityManager` class. The `disableMdo()` method is in turn called by three other methods within the same class. All together, the call chain introduces five valid framework entry points to the target `IUrspManager.setUrspBlackListUidRule(...)`. Apps can call any of them to trigger the target. To correctly detect Residuals, our analysis recovers all entry points for each API instance through a specialized backward search over the framework classes. More details are in Section 4.7.1.

This technique guarantees that we do not miss a target's active usage points within the device. It also avoids wrongly flagging certain APIs as Residuals even if we cannot identify an active site. Specifically, observe in the above example that `ConnectivityManager.requestNetwork(...)` is a public Android SDK API, implying that the private API `IUrspManager.setUrspBlackListUidRule(...)` is designated by Samsung to be indirectly reachable to third-party apps. Obviously, even if our analysis does not spot a usage point, the API could still be invoked via the public SDK API by other third-party apps to be installed later. We leverage this observation to rule out inspecting custom APIs reachable through public SDK APIs from our analysis. Specifically, after recovering framework entry points for a target API, our analysis proceeds to identify its usage points only if it is not transitively reachable via a public SDK entry point. Thus, `IUrspManager.setUrspBlackListUidRule(...)` will be skipped.

Challenge 2: Recognizing Residual Patterns over Time/Models

Studying one Residual instance within the whole population may not reveal interesting properties. As such, we must clearly define our investigation scope to infer meaningful Residual access control properties. To this end, we formally group Residuals into two categories based on their usage patterns: (1) *Historical Residuals* denote APIs that were active in *older* models but have ceased being used in successor and new models while (2) *Model Residuals* reflect APIs that are exclusively active on select device models from various versions. We detect Historical Residuals by observing the usage history of the APIs and recognizing the ones retiring (gradually or abruptly) over time. In contrast, to detect Model Residuals, we cluster similar devices (at the series or model level) and identify model-specific usages regardless of the version.

In the next section, we describe our solution in detail and elaborate on how we solve the above challenges.

4.6 Overview

To investigate Residuals at large scale, we design and implement ReM[2], a set of new analysis techniques that detect and evaluate the risks of custom Residuals. In this section, we first present our high-level idea and then describe the details of the proposed techniques.

[2]ReM: Short for *REMNANT*

Architecture

ReM is composed of three components: a ROM Analyzer, Usage-Pattern Extractor and Risk Identifier. Given a set of custom APIs in a device, the ROM Analyzer identifies *Likely* Residuals through a synergy between framework and preloaded app analysis. At the framework layer, ReM exhaustively collects public entry points that transitively lead to the invocation of the custom APIs. Through preloaded app analysis, ReM identifies *live* usage sites leading to a custom API either directly by calling the API's Remote Procedure Call (RPC) point or indirectly by calling the identified entry points. Unused APIs are flagged as *Likely* Residuals.

The Usage-Pattern Extractor confirms *Actual* Residuals through a large-scale cross-ROM analysis. Specifically, the module identifies *Historical* Residuals by running the above analysis repeatedly over a pool of curated ROMs for a target vendor, ordered by release date. It similarly recognizes *Model* Residuals by running the analysis over a cluster of ROMs from the same model/series. It subsequently builds a usage history for each *Likely* Residual in an attempt to identify the ones conforming to *Actual* Residual patterns characterized by an abrupt or a gradual retirement over time, or reflecting a consistent model-specific use.

Finally, Residuals are handed over to the Risk Identifier, which performs specialized program analysis on the Residual implementations to uncover two potential flaws: (1) Unsound security feature use and (2) obsolete access control. To detect the former, it leverages a set of patterns indicating unsound feature use and looks statically for their presence. Examples of these patterns include the use of package checks without co-located configuration checks and the use of an undefined permission. To detect the latter, the Risk Identifier performs a highly-optimized inconsistency detection and accordingly infers anomalous obsolete access control enforcement. It finally reports vulnerable Residuals.

4.7 Automated Detection of Residuals in Custom ROMs

Given the sheer number of analysis targets (framework and system app classes) and the large number of ROMs required for the historical analysis, ReM's analysis must be scalable and efficient.

4.7.1 Identifying Likely Residuals in a ROM

ReM conducts program analysis of the framework and preloaded apps to detect unused custom APIs. It first identifies framework-level entry points leading to the custom APIs and then statically looks for usage sites leading to the invocation of the APIs or corresponding entry points.

As mentioned earlier, identifying framework entry points is important since OEM private APIs are often available to framework and system app developers through custom *Manager* APIs (e.g., `UrspManager.setUrspBlackListUidRule` in Figure 4.4). These *Manager* APIs may be transitively invoked by other internal framework and SDK methods, forming indirect call chains from the components to the custom APIs.

Collecting Framework Entry Points

We first use the static bytecode analyzer WALA to process the framework libraries and extract defined classes and methods. Now, performing a forward search on each method to extract reachable APIs may sound compelling. However, it is likely that it will encounter and analyze many irrelevant methods and code fragments, unarguably affecting the scalability of the overall detection. To tackle the issue, we propose a more focused approach. We start with our set of target custom APIs, and perform backward expansion to iteratively discover public calling methods. Specifically, we use WALA to perform a class hierarchy analysis of the extracted classes and methods. Then for each method, we perform a depth-first reachability analysis on its call graph and locate the occurrence of a target custom API. If the latter is located, the calling method is added to the set of the target API's callers and is transitively fed back to the analysis loop to locate its potential public callers. The backward exploration constructs a mapping between each custom API and its calling methods and stops once no public callers can be encountered. Since the call chains are inherently deep, we optimize the exploration by:

- Caching discovered caller-callee mappings. The exploration consults the cache before moving on to look for other callers in order to avoid duplicate path exploration.
- The exploration stops preemptively if a public SDK method is encountered. That is, if a caller matches the name of a public API (which we have compiled for each Android release), the target API is ruled out from further analysis since it can be invoked by third-party apps. We further rule out the public API's direct and transitive callees from subsequent analysis, essentially considering the whole call chain accessible to third-party apps.

Collecting Usage Points in Apps

In this task, we statically analyze the apps and internal framework classes to collect usage points of a target API. Specifically, for each app, we perform standard forward reachability analysis starting from the app's public entry points (Android component life cycle methods and callback methods) and search for invocations to the targets. The analysis looks for invocations to the API's exposed Binder method and to its extracted framework entry points. To optimize the exploration, the search prioritizes entry points at the top level of the recovered caller-callee mappings chain and skips looking for a callee if a caller has already been encountered. Our analysis further handles calls to the APIs through Java reflection. During the reachability analysis, we treat reflection call methods as potential sinks if the arguments match the API's recovered framework entries or the RPC method itself. Specifically, for each Java reflection call that allows method invocation, we perform string analysis to extract the value of the call parameters (class names and method names). We use constant propagation within an analyzed app's inter-procedural CFG to resolve the method name in a reflective call (e.g., method.invoke(object)) and the class name that the method belongs to. A string variable from external input is modeled by a special value that denotes any string. We note that we are not interested in resolving the type/values of the arguments passed. This is sufficient for most of the cases we encountered.

Collecting Usage Points in System Services

We further look for call sites to the target APIs in the system services classes. We note that triggering the system service functionality may be initiated by the system server itself (e.g., in init methods, inner methods not exposed through IPC, etc.) through non-traditional channels (e.g., from the native layer). Thus, we mark any API that is triggered on the server side as a used API. Observe that this approach is conservative and is likely to overestimate the usage sites of APIs since a recovered site might not be necessarily invoked (i.e., it might occur in a dead code area).

At this stage, identifying *Likely* Residuals is straightforward; unused custom APIs are flagged as *Likely* Residuals.

4.7.2 Characterizing and Confirming Residuals

As stated earlier, we categorize Residuals based on their usage patterns, as follows:

1. A likely Residual is a Historical Residual if it gradually or abruptly retires over time. That is, the API's usage pattern decreases over time, until it is no longer in use in new successor devices.
2. A likely Residual is a Model Residual if it is consistently used in specific device series and models but not in others.

Observe that the two categories are inherently overlapping, since Model Residuals may also become unused over time.

ROM Collection and Curation

To detect Historical and Model Residuals, we perform a broad analysis of 628 custom ROMs released over the last ∼10 years (from Oct 2011 to May 2021). These ROMs are representative of major mobile vendors. More details on the sample ROMs can be found in Section 4.9.

We curate the samples for our analysis by carefully considering the following three properties of a ROM: (1) vendor, (2) model and (3) release date. We construct a usage history for a given API by analyzing chronologically ordered ROMs produced by the same vendor. We similarly build model-specific usage by grouping ROMs from similar series and models.

To identify the properties, we process a ROM's `build.prop` file (containing device properties) and extract the values of `ro.product.brand`, `ro.product.model` and `ro.build.version.release`. Note that a few vendors customize these attributes so we had to treat them on a case-by-case basis.

Scope of Analysis

ReM builds the usage patterns of the likely Residuals by running a per-ROM analysis over our curated pools of samples. In total, our analysis involved inspecting 48,000 unique preloaded apps (more than 250,000 all together) and led to identifying 6,349 custom APIs that exhibit actual Residuals patterns. More details can be found in Section 4.9.3.

In the next section, we describe how we evaluate the detected Residuals' security properties.

4.8 Automated Security Evaluation of Custom Residuals

In this section, we evaluate Residual access control enforcement. Our focus is on *evolution-induced access control vulnerabilities* that arise when framework developers do not safeguard Residuals. We classify these vulnerabilities as either unsound security features or obsolete access control enforcement. Unsound security features include undefined and device-incompatible features. Obsolete access control occurs when Residuals are not maintained and their access control enforcement is not updated and strengthened along with non-Residual APIs.

4.8.1 Evaluation Scope

We note that both classes of evolution-induced access control vulnerabilities examined in our security evaluation result from the presence of unused functionality. We focus on these particular vulnerabilities since, intuitively: (1) unused functionality is likely to be overlooked during updates and model customization and (2) in many cases, unused functionality is not even intended for use on a target device. Other types of vulnerabilities – particularly those that are *equally likely to occur in used APIs*, such as improper input validations, are out of scope for our evaluation.

Next, we describe how ReM detects the two classes of evolution-induced access control vulnerabilities.

4.8.2 Unsound Security Features

As stated earlier, the correctness of access control enforcement heavily relies on the soundness of adopted security features. Certain features may imply different protections depending on the running device version and build characteristics. Thus, a sound feature on a device where an API is used might not be sound on other devices where the API is not used.

Undefined Custom Permissions and Broadcasts

Custom permissions and protected broadcasts are introduced by customization stakeholders to protect custom resources. They are added, removed and renamed frequently. Removing a custom permission is performed when the defining stakeholder is not involved in a particular customization or when the permission is not needed. Other custom permissions are introduced by vendors and are tightly related to hardware. They are debloated when the corresponding resource is considered *nonfunctional*. For example, Samsung may remove permissions required to access its `Pen` functionality if the device does not embed a physical pen hardware. Protected broadcast definitions are removed for similar reasons.

Removing custom permissions and protected broadcast definitions is largely fine when *all APIs* referencing them are simultaneously removed. However, in the case of unmaintained Residuals, the occurrence of such references is highly problematic. Using an undefined security feature is unsound. As reported by the study [24], any app that defines removed features can silently gain the privilege to access the components referencing them. (Refer to Section 4.4 for an example.) To detect this pattern, ReM performs the following analysis:

- For each reported Residual, ReM statically extracts its implemented access control enforcement and identifies used security features. Specifically, it first builds the Residual's inter-procedural Control Flow Graph (CFG) and traverses it to extract invocations to security-relevant APIs (e.g., `checkPermission`, `enforceCallingPermission`). It then traces back from the APIs and keeps track of the permission string constants passed as arguments. ReM similarly processes registration sites of framework-defined broadcast receivers to extract corresponding actions.
- For each ROM with Residual instances, ReM collects the definitions of security features by running an XML parser over the framework and preloaded apps' manifest files.
- Last, ReM conducts a differential analysis to pinpoint Residuals that use undefined security features.

Package Name Checks without a Co-Located Check

Using package names for access control enforcement is not always sound. Since the names are forgeable identifiers, any party can squat the property and pretend to be the caller. In the Residuals scenario, since the expected calling package does not exist, the property is forgeable. Nonetheless, the property may become sound if used in conjunction with other checks. As stated earlier, configuration checks can validate a package name check. Traditional checks such as signature checks and other persistently sound checks (e.g., UID checks) naturally strengthen package name checks.

To detect the use of unsound package name checks, ReM traverses a target Residual's interprocedural CFG to collect invocations to the following: (1) APIs that retrieve the package name of the caller (e.g., `PackageManager.getNameforUid` and `PackageManager.getPackagesForUid`), (2) signature checks (e.g., `PackageManager.checkSignature`) and other checks for extracting the caller's unforgeable identifiers and (3) configuration checks. ReM then inspects the collections and marks sole invocations to package name checks as *potentially unsound*. Last, ReM verifies whether the target ROM does not include the specified package name to confirm unsoundness.

Resolving Strings

We observe through our analysis that package names returned from the `PackageManager` APIs (e.g., `getNameForUid`) are sometimes compared with dynamically constructed strings; i.e., by concatenating substrings, including constants, parameters and return values of other methods. We employ def-use analysis and examine if the package name returned from the target `PackageManager` APIs is compared with a string. We then use inter-procedural backward slicing and forward constant propagation to transitively resolve the strings. String arguments to other package check APIs (e.g., `PackageManager.getPackageUid`) may also be dynamically constructed and we resolve them similarly. We model strings that cannot be statically resolved (e.g., read from a framework resource file) with a placeholder that denotes any string. Our analysis conservatively considers a package name string that cannot be fully resolved to be sound.

Collecting Custom Configuration Checks

Besides using common AOSP APIs (e.g., `SystemProperties.get()` and global static fields (e.g., `OS.Build`), we observe through our analysis that vendors use a variety of custom methods for device configuration checks. Our inspection shows that these methods are often wrappers around AOSP APIs and usually involve multiple call chains. While performing inter-procedural CFG traversal will ultimately discover the underlying AOSP checks, it will encounter many irrelevant methods and affect the overall extraction performance. We tackle the issue by performing a one-time per vendor backward propagation (similar to the approach discussed in Section 4.7.1). The backward exploration builds a mapping between AOSP configuration check APIs and their calling methods, which we manually inspect to filter out custom configuration checks.

For each vendor, the automated backward propagation yielded 42 to 74 candidate configuration check methods. Our manual filtering yielded 19 to 24 actual configuration methods per vendor. We note that the manual filtering process is a small scale, one time effort.

4.8.3 References to Deprecated Security Features

For graceful removal of a security feature, framework developers may first flag it as deprecated, through the Java *@Deprecated* annotation. The deprecation subsequently pressures the developers to refactor their code and migrate to other alternative features. Eventually, after a few releases, the deprecated features are removed.

While the use of a deprecated security feature is not a vulnerability per se, we argue that it may eventually lead to one. Since Residuals are not used, they may not be properly maintained throughout version upgrades, leading to the persistence of deprecated security feature usage, even after the feature removal.

To detect this pattern, we use WALA to extract the Java annotations associated with the definition points of permissions and protected broadcasts (defined in the class `Manifest$permission`) and flag those annotated with `java.lang.Deprecated`.

4.8.4 Obsolete Access Control Enforcement

Android APIs are continuously evolving to add, fix and modify enforced access control. The evolution addresses new security requirements (e.g., migrating from a single-user to a multi-user device) and fixes reported flaws. A failure to keep up with the fast-paced evolution could induce obsolete access control enforcement, which may reflect weaker or absent access control enforcement.

Recognizing obsolete access control enforcement is not straightforward. Residuals implement custom functionality, with no publicly-available security specifications. As such, it is challenging to infer whether enforced access control is up-to-date. A popular approximate solution is to perform consistency analysis – essentially, comparing the access control enforced across multiple paths to the same resource and reporting inconsistencies; i.e., one path includes access control while the other does not. Various work exists in the area, ranging from approximate solutions [29, 31, 70] to more precise ones [21]. ReM follows an adapted version of the former approach since conducting a path-sensitive analysis will not scale to tackle the sheer number of APIs in our studied ROMs.

ReM conducts a largely-localized convergence analysis to identify other framework APIs that converge in functionality with the reported Residuals. It then extracts access control enforcement along the new APIs and compares them to those enforced by the Residuals. Observe that performing a framework-wide convergence analysis would not scale as some ROMs are extensively customized (e.g., more than 2000 custom APIs). To speed up the analysis, we limit our convergence analysis to (1) APIs defined within the same system service and (2) APIs defined in system services providing similar functionality. We leverage similar naming patterns to infer whether two system services provide overlapping functionality (e.g., *SemClipboard* and *Clipboard* services, *ISmsEx* and *ISMS* services).

To infer whether a Residual reflects updated access control, ReM further conducts a cross-ROM inconsistency analysis similar to [21]. Specifically, ReM compares the access control enforced by an API across multiple ROMs with different use scenarios; i.e., cases where the API is used in one but the Residual in the other.

We applied ReM to evaluate the access control enforcement of the Residuals identified in our ROM samples (i.e., 6,349 Residuals). ReM uncovered 1,453 violations. Details about the Residuals landscape and pertaining security properties are discussed next.

4.9 Large-Scale Measurement Study

To measure the pervasiveness of Residuals in the fragmented Android ecosystem and to understand the scope and magnitude of access-control anomalies they may pose, we perform a large-scale study of 628 ROMs.

4.9.1 Study Setup

The study has been conducted using 4 server machines equipped with 1/4 TB RAM, 16 cores, 64 Gbps net, 4 NVIDIA K10 GPU cards, each containing 2 GK104 GPUs.

4.9.2 Data Collection and Processing

Factory ROMs Collection

We collected 628 custom ROMs released over the last ∼10 years (from Oct 2011 to May 2021). The ROMs cover 7 major releases (from 4.0 to 10) and are customized by 7 vendors and cover 105 device models. We developed a crawler that automatically downloads vendor ROMs from public repositories. The crawler tries to cover as many distinct models and versions as possible to identify Historical and Model Residuals.

Table 4.1 lists the detailed statistics of our collected dataset. As shown, the ROMs are representative of big and medium players in the mobile market. We note that unlike Samsung and Blu ROMs, for which many dedicated public repositories are available, some vendor ROMs are more difficult to obtain and thus constitute smaller sample sizes in the dataset.

Table 4.1: Collected ROMs

EOM	**Statistics** (#)	**API Level / Version Numbers**						
		19 4.4 - 4.4.4	**21-22 5.0 - 5.1.1**	**23 6.0 – 6.0.1**	**24-25 7.0 - 7.1.2**	**26-27 8.0 - 8.1**	**28 9**	**29 10**
Samsung	ROMs	16	23	61	48	52	116	49
	Models	13	22	29	33	15	32	17
	APIs	3482	2462	4273	3588	3454	2282	2386
	Apps	168	257	308	310	315	335	345
Blu	ROMs	16	14	31	14	3	2	2
	Models	12	11	26	11	3	2	1
	APIs	403	516	582	562	636	476	794
	Apps	107	108	99	109	97	122	101
LG	ROMs	6	7	5	4	3	4	4
	Models	3	3	4	3	2	3	3
	APIs	1352	1017	875	1422	1101	902	896
	Apps	140	151	104	159	214	202	237
Xiaomi	ROMs	3	2	4	2	2	5	3
	Models	3	2	4	2	2	4	3
	APIs	773	962	1033	714	771	589	539
	Apps	133	154	181	182	182	207	197
Huawei	ROMs	17	1	2	2	2	3	7
	Models	13	1	2	2	2	1	1
	APIs	1233	157	963	576	725	461	286
	Apps	109	115	119	89	97	93	143
Lenovo	ROMs	38	11	6	2	2	4	1
	Models	31	8	4	2	2	3	1
	APIs	1651	1375	990	820	556	373	199
	Apps	121	131	135	113	156	143	111
Asus	ROMs	3	3	2	5	2	2	2
	Models	2	2	1	3	2	2	2
	APIs	1064	773	948	892	599	364	89
	Apps	147	167	131	161	179	176	156

Framework, Preloaded Apps and Configuration File Extraction

We preprocess the ROMs to extract framework classes and preloaded apps. To do this, we first locate the system partition, which contains the relevant classes.

For Android versions 4 through 9, we search for the system partition within `system.img` or `system.img.ext4` files. For LG images, the firmware is often packaged in a `kdz` file. To extract the system partition from `kdz` files, a modified version of the `SALT` tool [10] is used to generate `dz` files and to extract the embedded system partition (`system.image`). Once the system partition has been located, `imjtool`[7] is used to extract the image file, which is then mounted. When working with Android 10 ROMs, the system partition identification process differs slightly. Android 10 introduces the dynamic partitioning system, which allows partitions to be resized, created and removed during over-the-air updates [6]. As a dynamic partition, the system partition is housed within a larger super partition (`super.img`). To unpack the super partition, the sparse image is first converted to a raw image using `simg2img` [11]. Then we unpack the raw image using the `lpunpack`[8] tool, obtain a `system.img` file and proceed to mount the system partition.

We then extract each mounted image's build properties, specified in `build.prop` files. We then extract any framework or app `odex`, `vdex`, and `apk` files and use the `vdexextractor` [12], `baksmali`[5], `smali`[5], `apktool`[4], and `oat2dex`[9] tools to generate `dex` and manifest files.

4.9.3 Analysis Complexity

Codebases

Our analysis investigated 26,883 *distinct* private APIs and 48,000 preloaded apps. Table 4.1 presents the detailed statistics. The third row of each OEM entry lists the average number of *private* APIs recovered by ReM (not including AOSP's exposed APIs). As shown, vendors introduce 880 APIs, and the extent of customization differs between OEMs – with Samsung and LG exhibiting significantly more private APIs. As further illustrated in the fourth row of each vendor, the number of preloaded apps increases between major releases: it is larger in the latest releases. Observe that the results are reported as averages; some vendor-specific models include less preloaded apps than others (e.g., Samsung A/J Core models include ∼30% less apps compared to ZFlip models).

Recovered Entry Points

As discussed in Section 4.7, ReM collects framework entry points leading to custom APIs to accurately detect Residuals. Figure 4.5 reports the distribution of the recovered entry points, per OEM. For all OEMs, 50% of the APIs have 1 to 2 entry points; 25% have no entry points (meaning that the API is solely invoked via its RPC entry); and 25% exhibit a significantly larger number reaching up to 31 for Xiaomi. We investigated a few randomly selected samples that fall in the last category and found they often corresponded to methods for accessing custom information, e.g., custom profile information, whitelists for different services and keyguard information. Clearly, performing a simpler analysis that relies only on the RPC entry points and direct managers is likely to generate inaccurate Residual estimations.

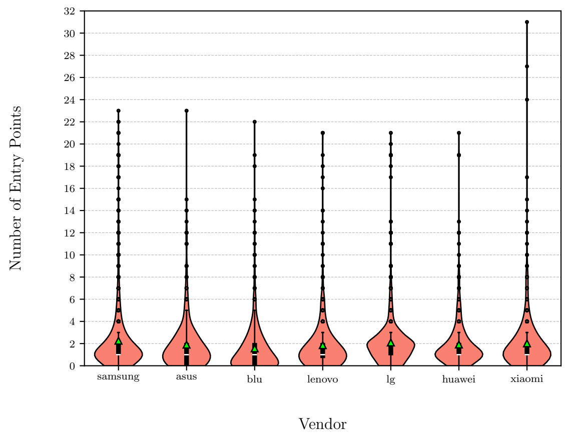


Figure 4.5: Distribution of the # of Entry Points per vendor

4.9.4 Residuals Landscape

Among all the 26,883 extracted private APIs, ReM discovered 6,349 instances that are Residuals in specific models/series or at specific release versions. We reiterate that as per our Model and Historical Residual detection, a used API is only flagged as a Residual if it exhibits certain trends (refer to Section 4.5).

Figure 4.6 depicts a breakdown of the reported Residuals per OEM. As shown, Residuals are prevalent among all vendors, reaching up to 42% in LG and Huawei (major releases 8.0 and 9, respectively). Blu ROMs exhibit the lowest number of Residuals since they are the least customized (i.e., smaller number of private APIs). We further note that Lenovo records 2% Residuals in version 10 because it was the least customized out of all the Lenovo samples.

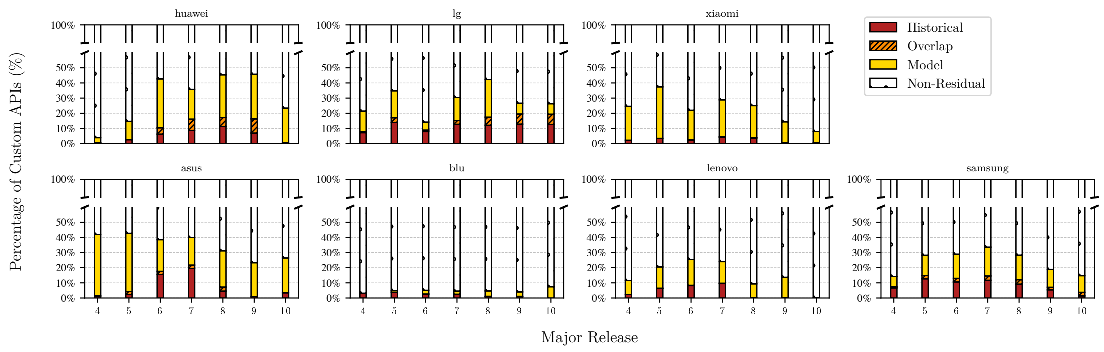


Figure 4.6: Residuals Breakdown

Observe that the number of Historical Residuals is lower in version 4, since the analysis cannot pick up the usage trend yet, as no data is available for earlier releases. The analysis only reflects the number of Residuals that are persistently unused on all releases.

Analysis Accuracy

From all the reported Residuals, we randomly sampled 50 and manually analyzed their usage in the corresponding ROMs (7 ROMs). We employed a simple word lookup to identify references to the Residuals (using the `grep` utility) in the preloaded apps and further investigated the references to verify if they were actively used (i.e., not included in dead code). We note that this analysis is simplistic since it is difficult to verify if a code region is dead, especially in the case of long call chains and obfuscated apps. Out of 50 instances, only 4(8%) were found to be false Residuals; that is, falsely reported to be not used while they were actually used in preloaded apps. Looking into these positives, we found that they occurred in obfuscated apps.

To estimate missed Residuals, we have similarly sampled 70 reported non-Residuals and manually analyzed their usage in the corresponding ROMs (11 ROMs). In all 70 samples, 5 (7.14%) were missed by ReM; i.e., Residuals considered to be used. We investigated these cases and found that they are caused by ReM's reflection handling (see Section 4.7.1). Since we do not resolve the type/values of the arguments passed in Java reflective calls, ReM cannot distinguish overloaded methods. Other cases were, due to infeasible code paths, conservatively treated by ReM as feasible.

4.9.5 Residual Lifespans

Figure 4.7 displays violin plots representing the distributions of the active and Residual lifespans for each OEM's Residuals. The active lifespan is the total number of versions a Residual is being actively used by some framework service or preloaded app, while the Residual lifespan is the total number of versions from a Residual API's introduction to its complete disappearance. Problems arise when an active lifespan is shorter than its corresponding Residual lifespan, as is the case for most vendors depicted in Figure 4.7.

The density of each violin plot corresponds to the frequency that a given lifespan is present in the larger population of active lifespans or Residual lifespans. We can see that, consistently, in overlapping regions between Residual and active lifespan distributions, the density is much higher for the Residual lifespan distributions.

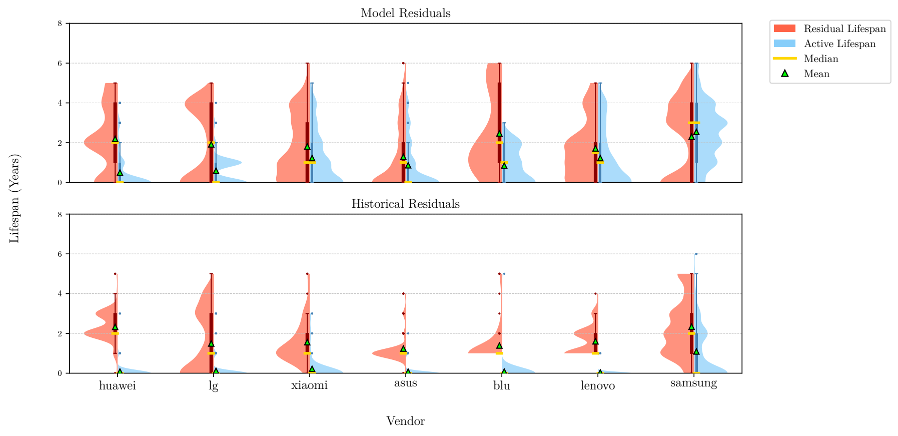


Figure 4.7: Violin Distribution of the Active and Residual Life Spans

For both Model and Historical Residuals, we can further spot that the mean active lifespan is almost always lower than the mean Residual lifespan. Note that a mean lifespan of zero implies that our analysis, spanning versions 4-10, did not identify any ROM instance actively using the Residual.

4.9.6 New versus Inherited Residuals

As shown in Figure 4.6, the percentage of Residuals is higher in versions 7-8 and starts a downward trend in versions 9 and 10. Although this signals that vendors are debloating

Residual APIs more notably in newer versions, the issue of Residuals is still prevalent among new versions. As shown in the Figure, Residuals exist in significant proportions in the latest versions; for example, LG, Huawei and Asus record between 23 and 28% Residuals in version 10.

To further demonstrate the importance of the Residuals issue in recent ROMs, we report the percentage of newly-introduced versus inherited Residuals throughout each new release. Figure 4.8 depicts the results; note that the results are aggregated for all vendors (per version).

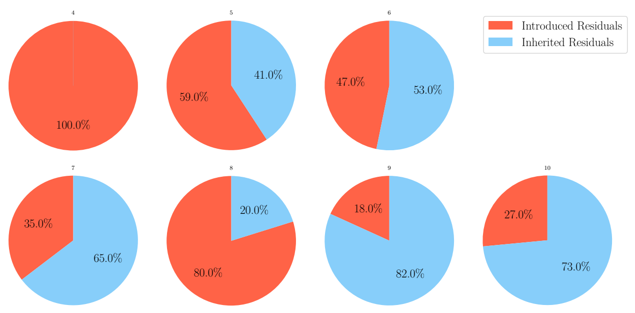


Figure 4.8: Inherited vs Introduced Residuals

As shown, 27% of Residuals are newly-introduced; i.e., they were active in version 9. This experiment clearly demonstrates that Residuals are not an issue of the past. This observation has yet to come to the vendors' full attention.

4.10 Residuals Security Landscape

In this section, we answer the following research questions:

- RQ1: Do vendor developers adopt *sound*, *compatible* security features while enforcing access control checks in Residuals?
- RQ2: Do vendor developers propagate *up-to-date*, *consistent* access control enforcement in Residuals throughout version upgrades?

Table 4.2: Unsound Security Features Use

Vendor	**Undefined Permissions**		**Deprecated Permission**		**Unsound Package Checks**		**Obsolete Access Control**	
	Residual APIs	Used APIs	Residual APIs	Used APIs	Residual APIs	Used APIs	Residual APIs	Used APIs
Samsung	273	19	229	90	339	188	402	113
Blu	2	0	0	0	23	6	14	9
LG	6	0	2	1	29	13	102	37
Xiaomi	0	0	0	0	26	10	48	25
Asus	8	0	0	0	18	7	31	16
Lenovo	12	5	0	0	4	7	33	21
Huawei	0	0	0	4	7	11	21	15

4.10.1 Unsound Security Features

Among all OEM Residual instances, ReM identified 978 flaws caused by the use of unsound security features. Observe that some of these flaws can be attributed to the same property (e.g., an OEM may use an undefined permission in three distinct Residuals, thus introducing three flaws). We also note that, although less common, multiple flaws may occur within the same API ($<\sim$ 3%).

Columns 2-7 in Table 4.2 depict a breakdown of unsound security features use per OEM. As listed, the number of flaws varies between vendors, with deprecated permissions being the least common in most vendors except for Samsung.

Undefined permissions are pervasive among Samsung samples. Examples include `com.samsung.accessory.manager permission.AUTHENTICATION_CONTROL`, `USE_LINK_TO_WINDOWS_REMOTE_APP_MODE` and `com.samsung.android.knox.permission.KNOX_EBILLING_NOMDM`, which ReM identified as causing more than 40 flaws in versions 9 and 10. All vendors used an unsound package check at the Residuals implementation. Examples include *com.sprint.**, *com.verizon** and **.docomo.**, which are left over from carrier-specific models. Lenovo and Huawei have the least flaws.

We note that the majority of our findings are spotted in Samsung largely because of its sample size (our collection includes more than 49 Samsung models as opposed to an average of 4 in other vendors).

4.10.2 Obsolete Access Control Enforcement

Columns 8-9 in Table 4.2 report the results of our conducted inconsistency analysis. As depicted, OEM Residuals do induce anomalies. ReM reported 14 to 442 inconsistency instances (505 all together), caused by the Residual leveraging a *different* security check to protect its underlying resources. We have inspected the results and confirmed that a significant proportion ($\sim 67\%$) exist due to OEMs overlooking the integration of *User* and *AppOps* checks. For example, LG adds 8 Historical Residuals in its custom *ISms* service which allows the handling of SMS functionalities without enforcing *AppOps* operation checks.

4.10.3 Comparison with Non-Residual APIs

Prevalence of Flaws among Non-Residuals

Evolution-induced anomalies may also occur in non-Residual APIs. Nonetheless, in contrast to Residuals, active APIs are better maintained and often undergo extensive security testing. To demonstrate that evolution induced flaws are less common in non-Residuals, we evaluate them using ReM. In Table 4.2, columns 3, 5 and 7 report the prevalence of unsound access control features and column 9 reports the number of detected inconsistencies. With the sole exception of Huawei's use of deprecated permissions in four non-Residual APIs, the flaws are significantly more prevalent in Residuals. Figure 4.9 depicts a breakdown of the flaws. As shown, Residuals are responsible for most of the reported vulnerabilities.

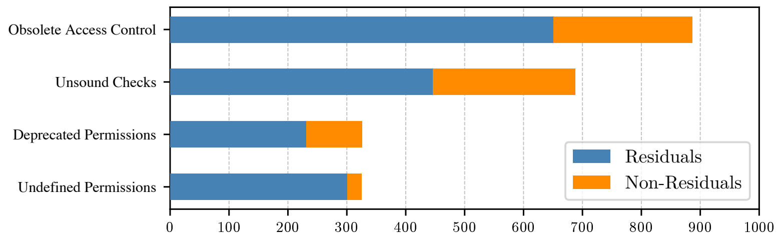


Figure 4.9: Flaws Breakdown in Residual and Active APIs

Comparison of Access Control Updates

To demonstrate that vendors may overlook updating Residuals in comparison to active APIs, we perform another experiment. For each custom API, we approximate its received access-control related updates as follows: we build a history of its adopted access control enforcement over time and report the number of observed distinct checks. We then compare the estimated numbers for Residual and non-Residual instances. Figure 4.10 reports the results. Both Historical and Model Residuals tend to receive less updates than Active APIs.

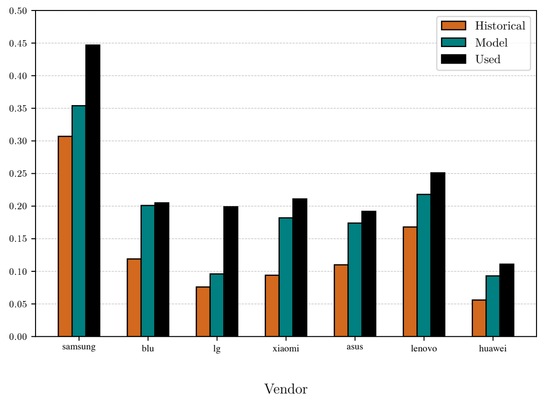


Figure 4.10: Average API Access Control Updates

4.11 Exploiting Residuals

We note that *not every Residual is exploitable.* Clearly, just like any other Android API, a Residual is exploitable depending on its provided functionality (e.g., a Residual that provides less sensitive operations may not be exploitable). Nonetheless, a privileged Residual API can open the door for exploits. While the ideal fix for a Residual is through its removal, it can be protected by a *strong* access control requirement or by a persistent non-configurable device property. However, if the proper protections are not in place, a Residual can be exploited to achieve security damages.

4.11.1 End-to-end POCs

To understand the security issues Residuals may pose, we analyzed a small subset of the reported weakly protected instances (93 cases). Our selection of the targets was based on the following three criteria: (1) comprehensibility of the Residual code, i.e., we avoided instances referring to proprietary functionalities with no public description; (2) availability of physical devices (specifically, LG and Samsung) and (3) sensitivity of operations – we prioritized sensitive APIs. Our manual analysis confirmed 8 exploitable Residuals. A summary of the findings is presented in Table 4.3. Note that, though the exploits span different devices as reported by ReM, we are conservatively listing here only the devices on which the attacks were manually confirmed. We have reported our findings to LG and Samsung. 7/8 have been acknowledged and fixed. One instance was marked as duplicate. Next, we describe a few instances.

Table 4.3: Confirmed Exploitable Residuals

Vendor	**Model**	**Residual Location**	**Impact**	**Vendor Reaction**	**CVE**	**NIST Ranking***
Samsung	S9	InputMethodManager	Corrupt Service Manager Device Shutdown	Confirmed, Fixed	CVE-2018-21088**	High (7.5)
Samsung	S10	SPENGesture	Keylogger	Confirmed, Fixed	CVE-2019-20597	Critical (9.1)
Samsung	S9	PersonaManager	Alter OEM Lock configurations Disable Keyguard Features Alter Profile Restrictions	Confirmed, Fixed	CVE-2020-25055	Critical (9.8)
LG	LG Q6	WindowManager	Keylogger	Confirmed, Fixed	CVE-2020-12754	High (7.8)
LG	LG Q6	Isms	Insert data into system providers	Confirmed, Fixed	CVE-2021-30162	High (7.1)
Samsung	J2/ A2 Core	InputManager	Keylogger	Confirmed, Fixed	To be issued	–
Samsung	S6 Note	PersonaManager	Launch activities through the system	Confirmed, Duplicate	–	–
LG	LG Q Stylo 4	IPhoneSubInfo	Read phone IMEI	Confirmed, Not Fixing	–	–

*The severity metric is reported based on CVSS 3.x.
**We note that we re-discovered CVE-2018-21088 using our tool. The issue was initially discovered by us manually.

Injecting Data into Privileged Content Providers

Our historical analysis of the LG samples reveals another major vulnerable Residual. The victim API `ISms.insertDBForLGMessage(...)` is defined in all LG devices running 4.4.4 up to version 10 [3] but is only used up to version 8.0 – thus becoming a Residual in versions 9 and 10. The Security analysis module reveals that it enforces obsolete access control – it requires the permission `android.permission.RECEIVE_SMS` while another path enforces a System check. A further dive into the Residual's implementation reveals that it allows inserting data to *any Telephony-accessible* content providers, while solely enforcing the aforementioned permission. Specifically, the Residual takes as arguments a Uniform Resource Identifier (URI) along with content values and then inserts the supplied values into the URI. Since the defining service `ISms` runs within the content of the Telephony process, the Residual can be exploited to insert data to any privileged provider that the process has access to – e.g., `Settings.Secure` and `Settings.System` providers, which maintain secure/system preferences that apps can read but not write. We confirmed the

[3]The API may have been introduced before version 4.4.4.

vulnerability through a PoC that targets `Settings.Secure` content provider to automatically replace the default IME with our specified IME (e.g., containing malicious keylogging functionality). LG has acknowledged and fixed the vulnerability. It is worth noting that, as confirmed by LG, the fix for Android R entails *removing the API*.

Keylogger on LG

We have identified another Residual `IWindowManager.setInputFilter(.)`. on LG Q6 that exhibits a similar pattern to the previous example. The Residual is defined on a few LG ROMs from versions 4.4.4 to 10 but is only used up to version 8.0. The Residual allows intercepting and controlling all input events before they are dispatched to the system or apps by registering an input filter. Alarmingly, our security module flagged the case as using an *unsound* security feature. Specifically, the API verifies if the calling package matches one of the two names: `"com.lge.systemserver"` or `"com.lge.onehandcontroller"`, and accordingly allows access to the filter registration. However, the API does not include any other checks – i.e., no configuration or signature checks. The historical analysis revealed that the above package names were indeed preloaded on the older devices and corresponded to the users of the API. However, in later versions, `"com.lge.onehandcontroller"` was removed, leaving the first path open to exploit. Observe that the other package, `"com.lge.remserver,"` persisted in the later version but did not invoke the target API. We have confirmed that the Residual can be exploited to build a keylogger by simply squatting the removed package name. LG acknowledged and fixed this issue.

Keylogger on Samsung

We discovered through our historical analysis that Samsung has introduces an API `ISpen GestureService.getCurrentInputContext(...)` in 27 ROMs starting from version 7.0 through version 8.1. Our cross-model analysis revealed though that the API is used only by 8 ROMs; all from SM-N95x and SM-T82x series (corresponding to SNote and STab devices). Consequently, the API was flagged as a Residual in the rest of the 19 ROMs. We have manually investigated this case and found that the API can obtain an instance to an `IInputContext` object, maintained by the defining system service (i.e., `SpenGestureService`). `IInputContext` abstracts the input method to an app and allows reading, editing and controlling user inputs such as taps and hard key presses. Given these privileged operations, obtaining this object is restricted to the system and input method managers in other framework call sites. Our security analysis module revealed the Residual has no security checks at all, allowing any third-party app to get the `IInputContext` object with no permissions. We have confirmed that the Residual can be exploited to intercept all user input including lock screen passwords, payment data and app credentials. We have further confirmed that it can be exploited to inject and compromise the integrity of user inputs. Samsung confirmed and fixed the vulnerability. NIST ranked the vulnerability as critical.

Launching Activities with System Privilege

Our historical analysis discovered the presence of a Residual instance in the majority of our collected Samsung samples (versions 8.0 through 10). The API `ISemPersonaManager.startActivityThroughPersona(..)` was introduced and exclusively used in earlier Samsung devices running version 7.0. Our security analysis flagged the case as potentially vulnerable since it enforced obsolete access control. We inspected the Residual and surprisingly found that it allows starting any Android activity within the highly-privileged context of the defining system service (named `Persona`). Specifically, it takes as an argument any arbitrary intent describing the activity to be launched and invokes Android's `Context.startActivity()` to trigger the specified intent. This is clearly alarming since it can be exploited to trigger system activities without a privilege requirement.

We have built an end-to-end PoC for version 8.0 to demonstrate possible damages. For instance, we supplied an intent with action `"android.intent.action.ACTION_REQUEST_SHUTDOWN"` to trigger a system shutdown. In another instance, we crafted an intent to call emergency phone numbers (with an explicit destination to the package `"com.android.phone"` with data `"tel:911"`); all by exploiting the unnecessary Residual functionality.

Samsung marked this vulnerability as duplicate. The issue was previously reported and fixed.

4.11.2 Other Impacts

The impacts of Residuals are significant. Besides the end-to-end PoCs we built (Section 4.11.1), we randomly selected 250 reported weakly-protected Residuals and manually investigated potential consequences that could happen once they were exploited. We note that the instances here are randomly selected. We do not necessarily have a corresponding physical device, and the Residual implementation may correspond to undocumented proprietary functionalities. As such, all we could do is to statically inspect the code and *estimate* possible consequences once a Residual is invoked. Such an analysis may not be accurate, but it is still important for evaluating the impacts of weakly-protected Residuals that have never been investigated before. The results of our analysis are shown in Table 4.4. We group the possible impacts by category (first column) and give a few examples for each category (third column).

Table 4.4: Impacts of 250 Randomly-Selected Residuals

Impact	**Count**	**Examples**	**Cause**	**Vendor(s)**
Data leakage	23	Infer location	OAC	Samsung
		Get Mac address	OAC	Asus, Lenovo
		Read network variables	USF	Lenovo
		Infer running apps	USF	Huawei
Data pollution	18	Delete files under dir	USF	Xiaomi
		Delete cache files	OAC	Xiaomi
		Insert text message to ICC	OAC	Blu
DoS	29	Change subscription state	OAC	Xiaomi
		Deny SMS receipt	USF	Samsung
		Remount file system	OAC	Blu
Global setting manipulation	34	Change Wlan configuration	OAC	Xiaomi, Blu
		Change keyguard configuration	USF	Samsung
		Change audio output path	OAC	Xiaomi
		Change SMS parameters	USF	Blu, LG
Unclear – Undocumented features	79	Set Drx Mode	USF	Samsung
		Change cycle time	USF	Samsung
		Process AT Command	USF	Blu
		Infer ENDIP sample	OAC	LG
No Risk	67	–	–	–

OAC: Obsolete Access Control; USF: Undefined Security Feature

As shown in the table, 23 instances of Residuals can be exploited to expose (sensitive) user data. Particularly, we identified one instance that could be invoked to register a listener, allowing an attacker to receive notifications of location updates. Other analyzed Residuals (18) allow manipulating data, including deleting cached files and other files under a specific directory. Our analysis further reveals 29 instances that can cause DoS attacks. One identified instance causes the device to deny and drop received SMS text messages. Another instance can be used to deny access to the external directory. We further identified other Residuals instances (34) that can be used to manipulate global settings, including Wlan configurations and SMS parameters. We could not predict the effect of 79 Residuals since they corresponded to undocumented proprietary features, while 67 other instances did not seem to lead to a clear security impact. As mentioned earlier, just like other APIs, weakly protected Residuals are not exploitable unless they implement a privileged functionality.

Chapter 5

Poirot

5.1 Introduction

During our analysis of access control flaws in Residual APIs, we noticed many access control inconsistencies, which occur when one path to a sensitive resource requires stricter access control enforcement than another. Malicious third-party application developers can take advantage of such inconsistencies to access sensitive resources through the least-protected path.

As we explored existing existing access control inconsistency detection solutions, we found that the state-of-the-art inconsistency detection tools suffer from two main limitations. First, their underlying detection methodology is highly-simplistic, often leading to inaccurate output unless substantial heuristics are adopted. Specifically, the tools are founded on the assumption that two APIs converging on an instruction (i.e., field update, method invocation) are related and thus require similar protections. However, we note that the convergence point may be auxiliary to the general functionality and hence likely irrelevant to the enforced access control. Failing to discern the relevance of the convergence point leads to significant false positives.

Second, the tools rely only on a *reachability* analysis to link an API and its accessible resources to derive their access control. We observe though that Android resources are often connected via implicit relations that can be structural, semantic and data-flow related. For example, a data-flow between two resources may imply that they require similar protections. Similarly, a naming similarity between a protected API and a reachable resource could help us infer that the resource is *likely* to require the API's protection. Modeling these implicit relations can help uncover new inconsistencies.

We present a new approach that reconceptualizes the inconsistency detection problem to account for uncertainty. Instead of assuming precise associations between resources and access control (i.e., resource r requires p), our tool assumes probabilistic ones (i.e., resource r may require p with confidence c).

Specifically, our solution works as follows: we begin by statically analyzing each Android API to collect *basic access control facts* through path-sensitive analysis. The facts correlate a resource r in the API to a protection p, which is a set of conjoint security constraints based on detected control dependencies. Each unique correlation is then assigned prior probabilities, values indicating our degree of belief in the access control implication.

Finally, the probabilistic inference engine aggregates the statically-collected basic facts, observations and constraints to project a high confidence protection recommendation for a resource. Depending on the type and number of facts and observations, the inference sharpens the initial probabilities and suppresses uncertainties. The generated probabilistic protection recommendations can then naturally be leveraged to detect access control inconsistencies.

We have integrated our proposed static analysis and probabilistic inference into an analysis pipeline, which we name Poirot. Our evaluation of Poirot shows that it is effective in generating protection recommendations for resources exhibiting sufficient facts and observations. Poirot can successfully predict *normalized* protections equivalent to AOSP implemented protections with an accuracy up to 84%. Our evaluation further reveals that our approach is effective in detecting inconsistencies. We run Poirot to analyze three custom images from Amazon, Xiaomi and LG, and discovered 26 true inconsistencies. While some of these inconsistencies may be detected via existing approaches, we note that 10 were uniquely discovered by Poirot.

5.2 Organization

We begin by providing a discussion on the shortcomings of existing inconsistency detection tools in Section 5.3. From Sections 5.4 to 5.7, we discuss the inner workings of Poirot in detail. Finally, we present our evaluation of Poirot in Section 5.8 and a case study of an inconsistency detected by Poirot in Section 5.9.

5.3 Limitations of Existing Inconsistency Detection Tools

While existing inconsistency detection tools have helped identify and correct significant access control anomalies, they suffer from two major limitations:

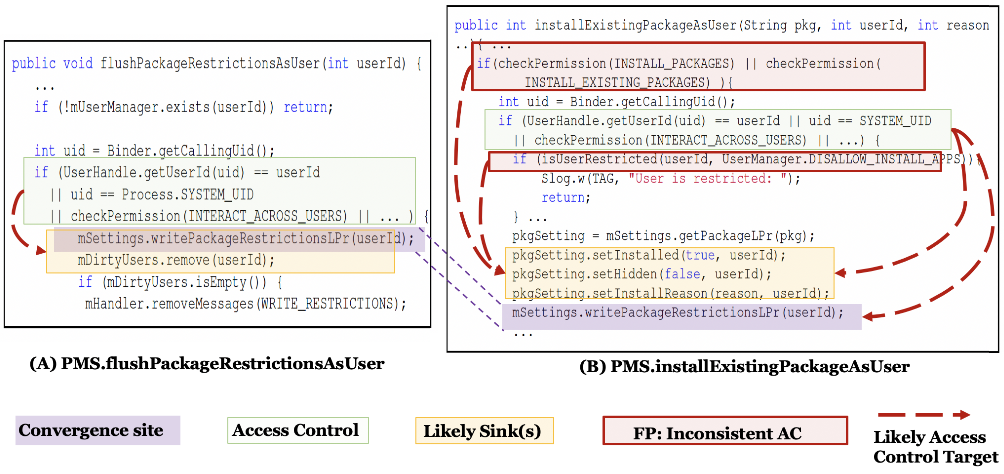
1. **Inaccurate Identification of Access Control Targets.** Since they may not accurately identify the targets of a given access control check, the existing tools generate *an overwhelming number of false positives.*
2. **Failure to Identify Implicit Access Control Inconsistencies.** As they can only detect explicit *reachability-based* inconsistencies, they may miss a significant number of *implicit* inconsistencies.

In the following subsections, we provide examples to illustrate both shortcomings.

5.3.1 Inaccurate Identification of Access Control Targets

Existing inconsistency detection tools consider two APIs to overlap in functionality if they converge on a similar instruction – for instance, if they invoke the same method or update the same variable. We refer to the similar instruction as the *convergence point.* If such a convergence exists, the tools examine and compare the enforced access control along the two paths from each API's entry to the convergence point and check if they are consistent. Essentially, the tools assume that the operation indicated by the convergence point should require *all* security checks found along with the most stringent access control path. However, this assumption is fundamentally inaccurate: the *convergence point may not be the target of the access control check* along the two paths. In fact, APIs commonly converge on instructions that are irrelevant to the enforced access control check.

Let us consider the code snippets (A) and (B) in Figure 5.1, extracted from AOSP (version 12). The highly simplified snippets depict the implementation of two APIs in the PackageManagerService (hereafter abbreviated as PMS) that perform two different functionalities: (A) `PMS.flushPackageRestrictionsAsUser(..)` flushes a specified package's restrictions for a given user to disk, while (B) `PMS.installExistingPackageAsUser(..)` installs an existing package for a specified user. Given the varying sensitivity of the operations, the two APIs enforce different access control checks. (A) performs a user ownership/ privilege check (shown in green), while (B) enforces a signature permission check



```
public void flushPackageRestrictionsAsUser(int userId) {
   ...
   if (!mUserManager.exists(userId)) return;

   int uid = Binder.getCallingUid();
   if (UserHandle.getUserId(uid) == userId
     || uid == Process.SYSTEM_UID
     || checkPermission(INTERACT_ACROSS_USERS) || ... ) {
       mSettings.writePackageRestrictionsLPr(userId);
       mDirtyUsers.remove(userId);
       if (mDirtyUsers.isEmpty()) {
        mHandler.removeMessages(WRITE_RESTRICTIONS);
```

(A) PMS.flushPackageRestrictionsAsUser

```
public int installExistingPackageAsUser(String pkg, int userId, int reason
..){ ...
  if(checkPermission(INSTALL_PACKAGES) || checkPermission(
        INSTALL_EXISTING_PACKAGES) ){
    int uid = Binder.getCallingUid();
    if (UserHandle.getUserId(uid) == userId || uid == SYSTEM_UID
      || checkPermission(INTERACT_ACROSS_USERS) || ...) {
      if (isUserRestricted(userId, UserManager.DISALLOW_INSTALL_APPS)){
         Slog.w(TAG, "User is restricted: ");
         return;
      } ...
      pkgSetting = mSettings.getPackageLPr(pkg);
      pkgSetting.setInstalled(true, userId);
      pkgSetting.setHidden(false, userId);
      pkgSetting.setInstallReason(reason, userId);
      mSettings.writePackageRestrictionsLPr(userId);
      ...
```

(B) PMS.installExistingPackageAsUser

Figure 5.1: False Positive Due to Inaccurate Identification of Targets

(`INSTALL_PACKAGES` or `INSTALL_EXISTING_PACKAGES`, shown in red) in addition to the user ownership/ privilege checks. As further depicted, despite their dissimilar functionalities, the two APIs converge on an internal method invocation `mSettings.writePackageRestrictionsLPr`, prompting existing inconsistency detection tools to treat the APIs as related. The existing tools would proceed to *wrongly* flag the least protected path leading to the convergence point (in this case, the path starting from the entry of `flushPackageRestrictionsAsUser`) as a potential inconsistency since it does not enforce the checks depicted in red in (B).

This shortcoming in existing tools is due to the inability of simplistic inconsistency analysis to accurately pinpoint the target(s) of enforced access control checks. To demonstrate this point, we assess the likely target of the checks implemented by the two APIs:

- The user checks (in the green box) implemented in `PMS.flushPackageRestrictionsAsUser` *likely* target all operations shown in the yellow box, including the convergence point `writePackageRestrictionsLPr` since they are all related to flushing and writing restrictions based on the user parameter – as inferred from their name and parameter values. Observe that we are uncertain about the relevance of the operation `mHandler.removeMessages` to the user check.

- The permission checks `INSTALL_PACKAGES` and `INSTALL_EXISTING_PACKAGES` and the user restriction check `DISALLOW_INSTALL_APPS` (in red) in `PMS.installExistin-`

gPackage are likely targeting the methods `PkgSettings.setInstalled` and `PkgSettings.setInstallReason` since their names indicate that they pertain to package installation.

- The user checks in `PMS.installExistingPackage` in the green box are likely targeting all operations in the yellow boxes as well as `writePackageRestrictionsLPr` since they all perform operations based on the user parameter.

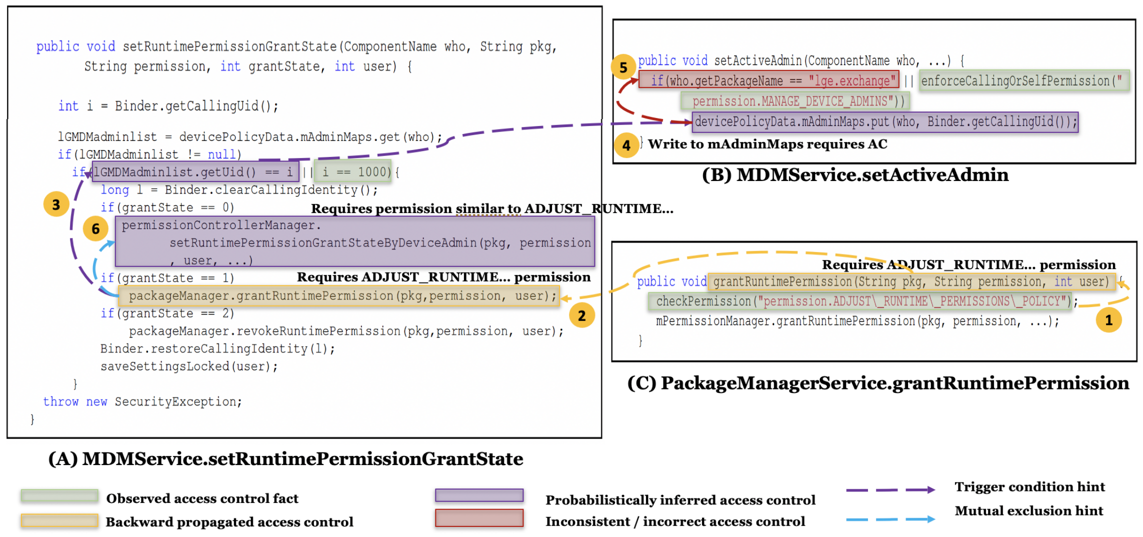
Based on this analysis, we deduce that the convergence point `writePackageRestrictionsLPr` is *highly unlikely* related to the permission checks required for package installation and to the user restriction check (`DISALLOW_INSTALL_APPS`). Hence, the detected inconsistency is a false positive. In practice, we have observed that this approximation results in a large number of false positives that overshadow true inconsistencies. We provide more details on the prevalence of false positives in Section 5.8.7).

5.3.2 Failure to Identify Implicit Access Control Inconsistencies

The previous work associates target resources with access control based on the notion of reachability, or whether a resource is reachable from a protected API. For example, in Figure 5.1(A), the resources `mSettings.write PackageRestrictionsLPr`, `mDirtyUsers.remove` and `mHandler.removeMessages` are all reachable from the API `flushPackageRestrictionsAsUser` and thus are assumed to require its protection – more specifically, the user checks in the green box. Note that control dependencies may be extracted to determine the right protection (as performed by AceDroid [21]). Reachability-based inconsistencies are then naturally detected if a resource is reachable from different paths exhibiting different protections. While reachability analysis can approximately associate a large number of resources with access control, we observe that *resources may also be linked to protections via other types of implicit relations*, including semantic, data-flow, and structural associations.

More importantly, we note that Android resources are usually transitively connected via more than a single relation. As reachability and convergence analyses cannot detect inconsistencies implied by such implicit and complex relations, they can overlook important inconsistencies.

To illustrate this, consider the motivating examples shown in Figure 5.2, extracted from LG V405E (version 10). Snippets (A) and (B) correspond to highly simplified implementations of two custom LG APIs defined in its `MDMService`. Snippet (C) depicts an excerpt from the AOSP API `PMS.grantRuntimePermission`. Linking access control



```
public void setRuntimePermissionGrantState(ComponentName who, String pkg,
      String permission, int grantState, int user) {

   int i = Binder.getCallingUid();

   lGMDMadminlist = devicePolicyData.mAdminMaps.get(who);
   if(lGMDMadminlist != null)
     if(lGMDMadminlist.getUid() == i || i == 1000){
       long l = Binder.clearCallingIdentity();
       if(grantState == 0)
       permissionControllerManager.
             setRuntimePermissionGrantStateByDeviceAdmin(pkg, permission
             , user, ...)
       if(grantState == 1)
         packageManager.grantRuntimePermission(pkg,permission, user);
       if(grantState == 2)
         packageManager.revokeRuntimePermission(pkg,permission, user);
       Binder.restoreCallingIdentity(l);
       saveSettingsLocked(user);
     }
  throw new SecurityException;
}
```

```
public void setActiveAdmin(ComponentName who, ...) {
  if(who.getPackageName == "lge.exchange" || enforceCallingOrSelfPermission("
      permission.MANAGE_DEVICE_ADMINS"))
    devicePolicyData.mAdminMaps.put(who, Binder.getCallingUid());
}
```

```
public void grantRuntimePermission(String pkg, String permission, int user) {
  checkPermission("permission.ADJUST\_RUNTIME\_PERMISSIONS\_POLICY");
  mPermissionManager.grantRuntimePermission(pkg, permission, ...);
}
```

1
2
3
4
5
6
Requires permission similar to ADJUST_RUNTIME...
Requires ADJUST_RUNTIME... permission
Write to mAdminMaps requires AC
Requires ADJUST_RUNTIME... permission
(A) MDMService.setRuntimePermissionGrantState
(B) MDMService.setActiveAdmin
(C) PackageManagerService.grantRuntimePermission
Observed access control fact
Backward propagated access control
Probabilistically inferred access control
Inconsistent / incorrect access control
Trigger condition hint
Mutual exclusion hint

Figure 5.2: Probabilistic inference of Access Control Checks and Implicit Inconsistencies

information pertaining to the resources in the three APIs reveals a (serious) implicit inconsistency in LG's `setActiveAdmin`. The inconsistency in (B) allows a third-party app to manipulate the content of `mAdminMaps`, a local resource used as a trigger condition in (B) `setRuntimePermissionGrantState` (highlighted in the purple box). Having full control of this important field allows the third-party app to subsequently trigger the underlying privileged operations (highlighted in yellow), including granting itself runtime permissions (via (C) `grantRuntimePermission`). Observe that this case would go undetected by existing inconsistency approaches since there is no clear reachability-based access control violation.

We are motivated by these types of implicit inconsistencies that require reasoning about various relations (e.g., structural, semantic, data flows) between resources and aggregating the pertaining access control information. We note that this reasoning entails a degree of uncertainty; we cannot be fully sure that an observed relation always implies a certain protection.

Returning to our example in Figure 5.2, we can infer the implicit inconsistency in API (B) by following the ordered steps: (1) statically analyzing snippet (C) shows that `grantRuntimePermission` requires the permission `ADJUST_RUNTIME_PERMISSION`. In (2), we propagate this information to the API's call site in `setRuntimePermi ssionGrantState`. This indicates that the latter should enforce a permission with a minimum protection level equivalent to `ADJUST_ RUNTIME_PERMISSION`. In (3), we observe that the API `setRuntime-PermissionGrantState` does not implement a permission check along the path leading to `grantRuntimePermission`; rather, it uses the trigger condition check pertaining to a read of the field `mAdminMaps` to control access. We refer to such a construct as *a trigger-condition hint*, indicating that the trigger *likely* provides a protection required by the reachable resource (i.e., `grantRuntime Permission`). Intuitively, this implies that the trigger should not be altered by a third-party app – unless it holds a permission equivalent to (or stronger than) `ADJUST_RUNTIME_PERMISSION`. In (4), we propagate the implied access control information to the write site of the field `mAdminMaps.put` in API (B). In (5), we detect a violation of this implication due to the flawed check in the red box.

By analyzing API (A), we can discern another *hint* that can help us reason about access control requirements. The boxes linked with a blue arrow indicate mutually exclusive operations that are preceded by a common trigger condition[1]. As such, we can infer that both operations are *likely* to require similar access control. This hint is particularly important when propagating the access control requirement extracted from `grantRuntimePermission` to LG's custom method `permissionControllerManager.setRunTime...`, as shown in (6). The mutually exclusive relation is effective in helping us derive the access control requirement for the custom resource and subsequently detect potential inconsistencies.

5.3.3 Inferring Implicit Inconsistencies

We are motivated by these types of subtle inconsistencies that require reasoning about various implicit relations between resources and aggregating the pertaining access control information. We note that this reasoning entails a degree of uncertainty; we cannot be fully sure that an observed relation always implies a certain protection.

Returning to our example in Figure 5.2, we can infer the implicit inconsistency in API (B) by following the ordered steps. In ❷, statically analyzing snippet (C) shows that the API `grantRuntimePermission` requires the permission `ADJUST_RUNTIME_PERMISSION`. In ❸, we propagate this information to the API's call site in `setRuntimePermissionGrantSt-ate`. This indicates that the latter should enforce a permission with a minimum protection level equivalent to `ADJUST_RUNTIME_PERMISSION`. In ❹, we observe that `setRuntimePermi-ssionGrantState` does not implement a permission check along the path leading to `grant-Runtime`; rather, it uses the trigger condition check pertaining to a read of the `mAdminMaps` field to control access. We refer to such a construct as a *a trigger-condition relation*, indicating that the trigger *likely* provides a protection required by the reachable resource (i.e. `grantRuntimePermission`). Intuitively, this implies that the trigger should not be altered

[1]Note that we do not consider input validations to be triggering conditions since they can be manipulated.

by a 3rd-party application unless it holds a permission equivalent to (or stronger than) `ADJUST_RUNTIME_PERMISSION`. In ❺, we propagate the implied access control information to the write site of the field `mAdminMaps.put` in API (B). In ❻, we detect a violation of this implication due to the flawed check in the red box.

By analyzing API (A), we discern another implicit relation that can help us reason about access control requirements. The boxes linked with a blue arrow indicate *mutually exclusive* operations that are preceded by a common trigger condition[2]. As such, we can infer that both operations are *likely* to require similar access control. This inference is particularly important when propagating the access control requirement extracted from `grantRuntimePermission` to LG's custom method `permissionControllerManager.set-RuntimePermissionGrantStateByDeviceAdmin`, as shown in ❻. The mutually exclusive relation is effective in helping us derive the access control requirement for the custom resource and subsequently detect potential inconsistencies.

Due to the *inherent uncertainties* in linking resources and protections, we observe that it is challenging to formulate general patterns that can precisely associate resources with access control. Statically extracted associations are imprecise for two main reasons. First, it is difficult to precisely pinpoint the resource(s) targeted by an observed access control check. Second, inferring an access control implication from an implicit observed association entails a degree of uncertainty. Consequently, it is difficult to accurately detect access control anomalies.

To meet these challenges, we propose a new solution centered around computing probabilistic protection recommendations for Android resources and leveraging those recommendations to derive potential inconsistencies. Our approach is based on the insight that the Android framework is rich with various *structural, semantic and data-flow hints* that link resources to protections and resources to other resources. These hints can be naturally consolidated into a protection recommendation using probabilistic inference. The probabilistic analysis will aggregate these hints and their frequencies to suppress uncertainties and infer protection recommendations.

5.4 Approach

Given an Android ROM, Poirot preprocesses the framework and system classes to identify system services and their APIs. It then statically analyzes the APIs to identify reach-

[2]Note that we do not consider input validations to be triggering conditions since they can be manipulated.

able resources and preceding access control checks in a path-sensitive fashion. Since the number of identified resources can prohibitively affect the probabilistic inference, Poirot statically preprocesses the APIs to eliminate irrelevant code blocks and reduce the number of resources to be further analyzed.

5.4.1 Basic Facts Collection

Poirot begins by collecting basic access control facts. Using an inter-procedural, path-sensitive analysis, the tool identifies possible paths leading to each resource in the reduced set. For each path, the tool extracts all enforced access control checks and considers them a conjoint set. It then introduces a random variable denoting the probability of the resource found at the end of the path to require the the conjoint set of access control checks. Observe that new random variables are added if the resource is found to require a new protection at other call sites.

5.4.2 Access Control Constraint Detection

For each resource, Poirot generates access control constraints, which assign prior probabilities to the random variables by analyzing access control properties – i.e., control dependency properties between resources and access control checks (regarded as basic facts). A prior probability is a value between 0 and 1 representing our degree of belief in a basic fact's access control implication. Particularly, a *one-to-one* control dependency between an access control check and a single resource is a strong indication that the resource is the target of the access control check. On the other hand, a *one-to-many* control dependency between an access control check and a set of resources implies that one or more items in the set is the likely target. As a result, *one-to-one hints* are more certain than *one-to-many hints*. Hence, we associate one-to-one hints with a 0.95 prior probability value while we associate one-to-many hints with a 0.60 prior probability value (More information on Poirot's prior probability values can be found in Section 5.8.3). Observe that the generated access control constraints may only assign initial protections to a subset of the sinks reachable from the API since not all will be linked to enforced access control via the collected basic facts.

Note that uncertain protection assignments will be suppressed as *more observations* are collected and more constraints are established during inference.

5.4.3 Implication Constraint Detection

Poirot propagates the initial probabilistic access control information to other resources through *implication constraints.* These types of constraints encode observed structural, semantic and data-flow relations *that connect one resource to another resource* with some degree of confidence. In such a way, basic access control facts can be propagated from resource to resource. We have identified seven types of implication constraint categories: Reachability, Triggering Condition, Mutual Exclusivity, Name Similarity, Getter-to-Setter, Data-Flow, and Parameter Flow constraints. An implication constraint relates two predicates as follows: $pred_1 \xrightarrow{pr} pred_2$ where pr denotes our confidence in $pred_1$ implying $pred_2$ to be true. Similar to the previous step, Poirot relies on static program analysis to extract the relations and to construct the pertaining implication constraints.

Inference We pass the collected probabilistic constraints into a probabilistic inference engine, which outputs final protection recommendations for framework APIs. Framework developers can then compare each generated recommendation with the corresponding API implementation to detect access control inconsistencies.

5.5 Access Control Constraints

Before collecting access control constraints from an API, we first perform a resource reduction using program analysis. We eliminate all resources within the API that are commonly used for sanitization checks, logging and metrics collection.

5.5.1 Definitions

To facilitate discussion, we introduce a few Android-specific definitions in Figure 5.3. We use *func* to denote a `Function`, which could be either an API (an exposed Android binder interface entry point) or an internal method (an unexposed method used internally by the system). An `Expression` e denotes a construct made up of variables, operators and method invocations that evaluates to a single value. An `Expression` may be related to a `Resource` r (e.g, `motionEvent.X=300`) or to a `Protection` p (e.g, `Binder.getCallingUid() == 1000`) or to others. We use s to denote a `Statement`, which represents a complete unit of execution. It corresponds to either to a sequence of statements or to code blocks along the true/false branches in conditional constructs.

Our analysis considers three types of resources: (1) a `FieldAccess`, denoted by f, (2) an `InternalMethod`, denoted by m and (3) an `API`, denoted by a. f is categorized based on the access type (read or write), while m and a are categorized as setters, getters or standard methods. We rely on a few static rules and naming conventions to perform this categorization.

Along each unique execution path from an API a, a resource r may be protected by a set of security checks. `Protection` p represents the conjunction of these security checks (e.g., `UserHandle.id= Owner` $\wedge$ `permission="Location"`). Note that we approximately model the `Protection` p required to invoke a by taking a union of all security checks along the protection path.

⟨`Function`⟩	*func*	::=	`<signature>` $\{\ s\ \}$
⟨`Expression`⟩	e	::=	$E(r) \mid E(p) \mid E($`others`$)$
⟨`Statement`⟩	s	::=	$s_1; s_2 \mid e \mid$ `if` $(e)\ \{\ s_t\ \}$
		$\mid$	`if` $(e)\ \{\ s_t\ \}$ `else` $\{\ s_f\ \}$
⟨`Protection`⟩	p	::=	$c_1 \wedge c_2 \wedge \cdots \wedge c_n$
⟨`Resource`⟩	r	::=	$f \mid m \mid a$
⟨`FieldAccess`⟩	f	::=	$f^{read} \mid f^{write}$
⟨`InternalMethod`⟩	m	::=	$m^{getter} \mid m^{setter} \mid m^{others}$
⟨`APICall`⟩	a	::=	$a^{getter} \mid a^{setter} \mid a^{others}$
⟨`SecurityConstraint`⟩	c		

Figure 5.3: A Simple Language for Android Functions

5.5.2 Basic Access Control Facts

As mentioned earlier, we rely on program analysis to collect basic access control facts from the reduced set of resources within an API. From the basic facts, we generate access control constraints, which assign an initial protection p to a resource r with some confidence c. To collect the basic facts, Poirot conducts a path-sensitive analysis since resources may be protected with disjunctive or conjunctive checks within an API. First, we perform a forward control-flow analysis on the API's interprocedural control flow graph (ICFG) and identify the conditional branches on which a target resource is control dependent. We then process the branches to infer access control patterns (for example, one operand in the predicate evaluating to an invocation of `Binder.getCallingUid()`) and extract other pertaining constraints using `DefUse` chains (e.g., operator, variables used in the operands). If multiple constraints are found along the same ICFG path leading to the target resource, the analysis merges them using a logical AND (implying conjoint checks). Conversely, if multiple ICFG paths are found to lead to the target resource, the analysis merges the in-path constraints for each ICFG path using a logical OR (implying disjoint checks). Observe that the latter scenario indicates that the target resource is reachable from different paths.

For each unique access control path leading to the target, Poirot introduces a new random variable denoting the probability that the target requires the union of constraints along the path.

5.5.3 Access Control Constraints

Once the initial access control facts are collected, Poirot generates *access control constraints*, which can take the form of either one-to-one or one-to-many control-dependency constraints.

1-to-1 Control-Dependency Constraints

One-to-one constraints connect a protection to a single resource. They are detected when an access control path is found to lead to one single resource. For example, Listing 5.1 shows that the permission check `MOUNT_UNMOUNT_FILESYSTEMS` precedes a single call to `MoveCallbacks.unregister(..)`. As such, we can intuitively link the permission to the invoke statement with high confidence.

```
public void unregisterMoveCallback(IPackageMoveObserver callback) {
   if(checkCallingPermission(permission.MOUNT_UNMOUNT_FILESYSTEMS == GRANTED)
      this.mMoveCallbacks.unregister(callback);
```

Listing 5.1: unregisterMoveCallback

To gather one-to-one constraints, Poirot performs a depth-first traversal of an API's ICFG and identifies the unique resources that are control-dependent on an identified protection. For each discovered one-to-one relationship between a resource r and a protection p, Poirot formulates an access control constraint, depicted by Rule R_1 in Table 5.2: $\mathit{AccessControl}(p, r, \texttt{SELF}) = \mathit{true}$ (0.95) with the random variable $\mathit{AccessControl}(p, r, \texttt{SELF})$ asserting that p is derived from a one-to-one control dependency. Figure 5.4 describes the meaning of *AccessControl*. Note that the third parameter denotes the propagation direction, which we will discuss shortly.

1-to-n Control-Dependency Constraints

These constraints are detected when an access control path leads to more than one resource along a unique ICFG path. They reflect the less certain scenario where it is challenging to pinpoint the exact protection target(s) without additional clues. (Refer to the motivating examples in Figures 5.1(A) and (B) for illustration.) Poirot formulates this access control constraint (depicted by R_2 in Table 5.2) for each pair of protection p and its control-dependent resources $r \in R$, as follows:

$\mathit{AccessControl}(p, r, \texttt{SELF}) = \mathit{true}$ (0.60), with random variable $\mathit{AccessControl}(p, r, \texttt{SELF})$ asserting that the protection p is derived from a 1-to-n control dependency.

5.6 Implication Constraints

Implication constraints do not directly link a resource with a protection. Instead, they link resources to one another by leveraging observed structural and semantic relations statically connecting the resources. As such, these constraints propagate protection recommendations across resources. Note that a propagated protection could be directly assigned by a access control constraint or iteratively deduced during probabilistic inference. More formally, implication constraints are presented as an implication from a prior-probability predicate to a posterior predicate or from one posterior predicate to another posterior predicate. Table 5.1 lists the observations (O_1 to O_7) that Poirot relies on to establish the implication constraints. Below, we discuss in detail each observation and corresponding implication constraint. We note that our tool is extensible so new constraints can always be added to refine the analysis.

5.6.1 Structural Constraints

These constraints are identified by considering the program structure. They allow us to encode the most commonly used structures that we have observed.

Reachability

Reachability forms the most basic structural constraint that can connect two resources. A resource r_1 is reachable from r_2 if r_2 is the direct caller of r_1. We establish reachability hints exclusively between an API r_{caller} and its reachable resources. In other words, we do not

Table 5.1: Fact and Observation Definition

ID	**Facts and Observations** (derived from static program analysis)
F_1	*ControlDependency*$(p, R=\{r_1, r_2, ..., r_n\})$: a set of resources (R) are control-dependent on protection p.
O_1	*Reachability*$(func, R=\{r_1, r_2, ..., r_n\})$: a set of resources (R) are reachable from the entrypoint of function *func*.
O_2	*SameBlock*(e_1, e_2): expressions e_1 and e_2 are within the same basic block.
O_3	*Contains*(s, e): the expression e is a part of the statement s.
O_4	*Dataflow*(e_1, e_2): there is a direct data-flow from the expression e_1 to e_2.
O_5	*Argument*$(func, e)$: the expression e is an argument of the function *func*.
O_6	*NameCorrelation*(r_1, r_2): the resources r_1 and r_2 have name correlation.
O_7	*InPath*(p, a, r): protection p is located in the path from API a to resource r.

consider internal method reachability hints since our analysis is interprocedural. To collect reachability hints, Poirot builds a call graph for each API and performs an inspection to identify direct $\langle API\text{-}r_{callee}\rangle$ relations. Transitive reachability constraints will be encoded during probabilistic inference.

An observed reachability between API and r_{callee} implies that we can propagate API's inferred protections to its reachable callees. However, we note that some of the inferred protections may already be encoded through control-dependency constraints. Consider Listing 5.2:

```
public void removeUser(int userId) {
   if( checkPermission(MANAGE_USERS) == GRANTED || ...)
      removeUserUnchecked(userId);
```

Listing 5.2: removeUser

The forward reachability hint between caller `removeUser` and callee `removeUserUnche-cked` should not propagate the caller's in-API protection requirements to the callee. As such, our implication constraints are tailored to account for the direction of the inferred protection. As we show in Figure 5.4, Poirot considers five directions. A `SELF` direction denotes the cases where the protection is derived from a basic fact within the API's implementation. A `FORWARD` direction denotes the cases where the protection is inferred from the API's call site.

For example, the call site of `removeUserUnchecked` enforces a protection. A `BACKWARD` direction denotes the opposite: a callee's protection is propagated back to its calling API.

AccessControl(p, r, d): the resource r is protected by the protection p, which is inferred along with the direction d.

$d \in \{$`FORWARD`, `BACKWARD`, `SELF`, `AGGREGATED`$\}$.

`SELF`:	directly derived from facts
`FORWARD`:	forward propagation, i.e., following the program's control flow
`BACKWARD`:	backward propagation, i.e., reversing the program's control flow
`-`:	direction-free propagation
`AGGREGATED`:	the aggregated result from the three aforementioned directions.

Figure 5.4: Defining the Random Variables

In this case, `removeUser`'s protection is propagated back to some other invoking API. A `-` direction denotes a direction-free propagation (we discuss this case in greater detail later on). Finally, an `AGGREGATED` protection represents the cases where a protection is an aggregated result of different protection directions.

Intuitively, a reachability implication constraint is bidirectional in the `FORWARD` and `BACKWARD` directions and its confidence is subject to the same 1-1 and 1-n control dependency constraints. However, we note subtle properties regarding the propagation direction that should be accounted for. Let us use Listing 5.3 to understand the properties.

```java
public void reportFailedPasswordAttempt(int userHandle) {
    if(checkPermission(BIND_DEVICE_ADMIN) == GRANTED){
        Binder.clearCallingIdentity();
        policy.mFailedPasswordAttempts++;
        if(policy.mFailedPasswordAttempts >= max))
            if (userHandle == UserHandle.USER_OWNER) {
                wipeDataLocked(wipeExtRequested, reason);
            } else {
                am.switchUser(UserHandle.USER_OWNER);
                mUserManager.removeUser(UserHandle)
```

Listing 5.3: reportFailedPasswordAttempt

Below, we explain each step Poirot takes to generate observations and constraints from Listings 5.2 and 5.3:

1. The API resources `Recovery.reboot`, `am.switchUser`, and `mUserManager.removeUser` are reachable from the API resource `reportFailedPasswordAttempt` (*O1*).

2. The InternalMethod resource `removeUserUnchecked` is reachable from API `mUserManager.removeUser` (*O1*).

3. The API `mUserManager.removeUser` is associated with permission `BIND_DEVICE_ADMIN` through a 1-n control dependency (*R2*).

4. The InternalMethod resource `removeUserUnchecked` is associated with the permission `MANAGE_USERS` through a 1-1 control dependency (*R1*).

From (2) and (3), Poirot establishes a forward reachability constraint (*R3*) to propagate the following:

5. *AccessControl*(BIND_DEVICE_ADMIN, `mUserManager.removeUser`, `FORWARD`) $\xrightarrow{0.95}$ *AccessControl*(`BIND_DEVICE_ADMIN`, `removeUserUnchecked`, `FORWARD`).

From (2) and (4), Poirot generates the following backward reachability constraint:

6. *AccessControl*(`MANAGE_USERS`, `removeUserUnchecked`, `BACKWARD`) $\xrightarrow{0.95}$ *AccessControl*(`MANAGE_USERS`, `UM.removeUser`, `BACKWARD`).

Similarly, from (1) and (6), Poirot derives the following backward reachability constraints:

7. *AccessControl*(p, `UM.removeUser`, `BACKWARD`) $\xrightarrow{0.6}$ *AccessControl*(p, `report.PasswordAttempt`, `BACKWARD`).

At this stage, the backward derived permission `MANAGE_USERS` for `reportFailedPasswordAttempt` from `UM.removeUser` can be *further propagated in forward fashion* to other reachable resources based on (1). However, we note that the propagation would likely cause incorrect protection inference. We address this potential inaccuracy by limiting this propagation to resources in the same block. Rule R_6 enforces this constraint with 0.6 confidence to model this inherent uncertainty.

Triggering Conditions

Here we rely on the conditional control flow construct `if` *Trigger Predicate* `then` r to correlate resources. This construct is common in Android APIs that deliver a promised functionality only when certain triggering conditions are satisfied. For example, an API that allows the caller to send an SMS message may only invoke the actual sending functionality when the mobile data is active. The triggers often reflect global system properties such as hardware features, running device state or local properties defined in the resource's scope (e.g., policy contains a value).

Table 5.2: Probabilistic Inference Rules

ID	Conditions*	Probabilistic Constraints
R_1	$\mathit{ControlDependency}(p, \{r\})$	$\mathit{AccessControl}(p, r, \texttt{SELF}) = \mathit{true}$ (0.95)
R_2	$\mathit{ControlDependency}(p, R) \wedge \lvert R\rvert > 1 \wedge r \in R$	$\mathit{AccessControl}(p, r, \texttt{SELF}) = \mathit{true}$ (0.60)
R_3	$\mathit{Reachability}(a, \{r\}) \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\}$	$\mathit{AccessControl}(p, a, d) \xrightarrow{0.95} \mathit{AccessControl}(p, r, \texttt{FORWARD})$
R_4	$\mathit{Reachability}(a, R) \wedge \lvert R\rvert > 1 \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\} \wedge r \in R$	$\mathit{AccessControl}(p, a, d) \xrightarrow{0.60} \mathit{AccessControl}(p, r, \texttt{FORWARD})$
R_5	$\mathit{Reachability}(a, R) \wedge r \in R \wedge d \in \{\texttt{BACKWARD}, \texttt{SELF}, \texttt{-}\}$	$\mathit{AccessControl}(p, r, d) \xrightarrow{0.60} \mathit{AccessControl}(p, a, \texttt{BACKWARD})$
R_6	$\mathit{SameBlock}(E(r_1), E(r_2))$	$\mathit{AccessControl}(p, r_1, \texttt{BACKWARD}) \xrightarrow{0.6} \mathit{AccessControl}(p, r_2, \texttt{FORWARD})$
R_7	$\mathit{NameCorrelation(a,\ r)} \wedge \mathit{Reachability}(a, \{r\}) \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\}$	$\mathit{AccessControl}(p, a, d) \xrightarrow{0.70} \mathit{AccessControl}(p, r, \texttt{FORWARD})$
R_8	$\mathit{NameCorrelation(a,\ r)} \wedge \mathit{Reachability}(a, \{r\}) \wedge \mathit{InPath}(p, a, r)$	$\mathit{AccessControl}(p, a, \texttt{BACKWARD}) \xrightarrow{0.70} \mathit{AccessControl}(p, r, \texttt{FORWARD})$
R_9	$\mathit{NameCorrelation(a,\ r)} \wedge \mathit{Reachability}(a, \{r\}) \wedge d \in \{\texttt{BACKWARD}, \texttt{SELF}, \texttt{-}\}$	$\mathit{AccessControl}(p, r, d) \xrightarrow{0.70} \mathit{AccessControl}(p, a, \texttt{BACKWARD})$
R_{10}	$\big(\exists s,\ s.t.\ s \equiv \texttt{if}\ (E(r_2^{read}))\ \{s_t\}\big) \wedge \mathit{Contains}(s_t, r_1) \wedge d \not\equiv \texttt{AGGREGATED}$	$\mathit{AccessControl}(p, r_1, d) \xrightarrow{0.85} \mathit{AccessControl}(p, r_2^{write}, \texttt{-})$
R_{11}	$\big(\exists s,\ s.t.\ s \equiv \texttt{if}\ (e)\ \{s_t\}\ \texttt{else}\ \{s_f\}\big) \wedge (e \equiv E(\texttt{INPUT\_CHK}) \vee e \equiv E(\texttt{SYS\_PROPERTY})) \wedge \mathit{Contains}(s_t, E(r_1)) \wedge \mathit{Contains}(s_f, E(r_2)) \wedge d \not\equiv \texttt{AGGREGATED} \wedge \mathit{NameCorrelation}(r_1, r_2)$	$(\mathit{AccessControl}(p, r_1, d) \xrightarrow{0.90} \mathit{AccessControl}(p, r_2, \texttt{-})) \wedge (\mathit{AccessControl}(p, r_2, d) \xrightarrow{0.90} \mathit{AccessControl}(p, r_1, \texttt{-}))$
R_{12}	$d \not\equiv \texttt{AGGREGATED}$	$\mathit{AccessControl}(p, m^{getter}, d) \xrightarrow{0.80} \mathit{AccessControl}(p, m^{setter}, \texttt{-})$
R_{13}	$d \not\equiv \texttt{AGGREGATED}$	$\mathit{AccessControl}(p, a^{getter}, d) \xrightarrow{0.80} \mathit{AccessControl}(p, a^{setter}, \texttt{-})$
R_{14}	$\mathit{Data\text{-}flow}(E(r_1), E(r_2))) \wedge d \not\equiv \texttt{AGGREGATED}$	$(\mathit{AccessControl}(p, r_1, d) \xrightarrow{0.80} \mathit{AccessControl}(p, r_2, \texttt{-})) \wedge (\mathit{AccessControl}(p, r_2, d) \xrightarrow{0.80} \mathit{AccessControl}(p, r_1, \texttt{-}))$
R_{15}	$\mathit{DataFlow}(e, E(r)) \wedge \mathit{Argument}(a, e) \wedge \mathit{Reachability}(a, r) \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\}$	$\mathit{AccessControl}(p, a, d) \xrightarrow{0.70} \mathit{AccessControl}(p, r, \texttt{FORWARD})$
R_{16}	$\mathit{DataFlow}(e, E(r)) \wedge \mathit{Argument}(a, e) \wedge \mathit{Reachability}(a, r) \wedge \mathit{InPath}(p, a, r)$	$\mathit{AccessControl}(p, a, \texttt{BACKWARD}) \xrightarrow{0.70} \mathit{AccessControl}(p, r, \texttt{FORWARD})$
R_{17}	$d \not\equiv \texttt{AGGREGATED}$	$\mathit{AccessControl}(p, r, d) \xrightarrow{1.00} \mathit{AccessControl}(p, m^{setter}, \texttt{AGGREGATED})$

*Each fact/observation is encoded with a unique ID. As such, the more facts/observations (of the same type) that Poirot derives, the higher the confidence assigned to the corresponding constraint. We elide the details in the table for simplicity.

We observe that *altering the triggers* is usually a protected operation that requires at least the same privilege as that of the invoked resource. Intuitively, this is essential to prevent triggering the sinks adversely in unsupported situations.

Poirot generates the following implication constraints to encode this observation, depicted in Rule R_{10} in Table 5.2. If a resource r_1 is control-dependent on an expression pertaining to a read of resource r_2 – i.e., r_2^{read}, Poirot adds a unidirectional trigger implication constraint between the two predicates:

$$AccessControl(p, r_1, d^{3}) \xrightarrow{0.85} AccessControl(p, r_2^{write}, \text{-})$$

Here, we adopt a relatively low confidence given the uncertainty of this observation. Note that this implication is not related to reachability and hence is a direction-free propagation.

Mutual Exclusivity

Here we rely on control flow constructs in the forms (1) `if` *Predicate* `then` $r1$ `else` $r2$ and (2) `if` *Predicate1* `then` $r1$ `elseif` *Predicate2* `then` $r2$ to correlate $r1$ and r_2. These constructs are commonly used in APIs that provide varied implementations for the same functionality depending on the running device properties. For instance, a `sendSMS` API may check if the device is a CDMA or GSM model to select the relevant SMS dispatcher method (e.g., $dispatchCDMA$ vs $dispatchGSM$). Note that the triggered methods are mutually exclusive and provide semantically similar functionality. We rely on this observation to speculate that two mutually exclusive operations may require similar protections.

To detect this pattern, Poirot focuses on the structure of the control flow branch. The triggering predicate(s) should be related to system properties or to input checks and the individually triggered paths should be semantically related. We leverage a simple naming similarity analysis to determine equivalence (akin to the similarity measure followed in Section 5.6.2). Note that the analysis avoids flagging error and validation checks, which commonly follow similar constructs.

Once two mutually exclusive operations r_1 and r_2 are detected, Poirot adds a bidirectional implication constraint as depicted by Rule R_{11} in Table 5.2:

$$\Big(AccessControl(p, r_1, d) \xrightarrow{0.90} AccessControl(p, r_2, \text{-})\Big) \wedge \Big(AccessControl(p, r_1, d) \xrightarrow{0.90} AccessControl(p, r_2, \text{-})\Big)$$

[3]We omit direction details for simplicity. More details can be found in Table 5.2.

5.6.2 Semantic Hints

Semantic hints capture dependencies that exist between resources based on naming information or operation semantics.

Name Correlation

Here, we rely on the observation that Android framework code contains a considerable amount of semantic information to support comprehensibility and development. APIs, internal methods, fields and other program elements often possess meaningful names. More importantly, related elements are often named similarly. That is, the names may share a root or substrings. We leverage this knowledge to link resources together and refine their protection probabilities. Specifically, given a set of resources R reachable from a protected API, Poirot identifies the subset of resources whose names are similar to the API and accordingly creates a naming correlation implication constraint. This constraint implies that the API's protections are likely to be required for any resource bearing a similar name.

Back to Listing 5.3, we can spot a naming similarity between API `reportFailedPassw-ordAttempt` and the field resource `policy.mFailedPasswordAttempt`. Hence, we can accordingly increase our confidence in the field access `policy.mFailedPasswordAttempt` to require `BIND_DEVICE_ADMIN`, which was initially assigned through a 1-n control-dependency constraint.

To calculate the naming similarity score between two resources a and r, Poirot relies on the DICE coefficient score [80]. It then establishes a naming correlation implication constraint between a and r if the dice coefficient is substantially high.

Poirot The constraint is depicted in Rule R_7:

$AccessControl(p, a, d) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$

where direction $d \in \{\texttt{FORWARD}, \texttt{SELF}\}$. We note that when the learning direction is `SELF` (i.e., p is derived within a's implementation via a basic fact), we enforce an additional condition: r should be control dependent on p to exclude protections that may be targeting different resources in different branches.

As the naming similarity constraint is bi-directional, we can backward propagate protections inferred for r to a as shown in Rule R_8, where direction $d \in \{\texttt{BACKWARD}, \texttt{SELF}, \texttt{-}\}$:

$AccessControl(p, r, d) \xrightarrow{0.70} AccessControl(p, a, \texttt{BACKWARD})$

Getter-to-Setter

Here, we rely on operation semantics to correlate resources. We focus on linking *getter* and *setter* resources (for both APIs and internal methods) to transfer their protections. This constraint is founded on the general observation that a mutate/set operation is *likely* to be at least as restrictive as a get operation. We note that this observation may not hold in all cases. For instance, consider the case where appending to a shared buffer is allowed, but reading is not. However, the inherent uncertainty in this constraint can be suppressed during probabilistic inference.

To collect $\langle r^{getter}, r^{setter} \rangle$ pairs, Poirot constructs the ICFG of each API and detects all return statements. It then resolves the object returned as follows. First, if the object resolves to a global field, Poirot inspects other APIs to identify corresponding setters. Second, if the object resolves to a return value of other methods, Poirot transitively analyzes them following the same procedure to resolve the actual object returned. The tool similarly looks for corresponding setters. We note that we rely on a few rules to identify field get and field set operations. Details are elided due to space constraints.

For each identified pair, we construct the following implication constraint (depicted by rules R_{12} and R_{13}), which propagates the getter's protections to the setter. Note that this constraint is unidirectional.

$$AccessControl(p, m^{getter}, d) \xrightarrow{0.80} AccessControl(p, m^{setter}, \texttt{-})$$

$$AccessControl(p, a^{getter}, d) \xrightarrow{0.80} AccessControl(p, a^{setter}, \texttt{-})$$

5.6.3 Data-Flow Hints

Data-flow constraints denote define-use associations across resources. They are particularly helpful when deriving protection requirements for a resource that has not been associated with any particular protection but is linked to other resources via define-use relations. Consider the highly simplified snippets from two APIs spotted in FireOS in Listing 5.4.

As shown, there is no high-confidence access control constraint that assigns a protection to the global resource `moveId`. However, we can infer its protection via the data-flow constraint in line 11, which connects the resource to `APM.readMoveData()` , which turns out to require a signature protection. Note that this can help us transitively infer a new protection for `info.moveId` (line 5) through another data-flow constraint.

```java
String moveId;
public MigrationInfo getMoveData() {
     if (checkPermissio("READ_MOUNT_DATA") == 0){
        MigrationInfo info = new MigrationInfo();
        info.moveId = moveId;
        info.moveStatus = moveStatus;
        return info;
public void moveData() {
   moveId = readMoveData();
```

Listing 5.4: getMoveData

Poirot collects data-flow constraints as follows. First, for each r_1 update operation (e.g., a direct assignment statement, an add operation on a Java class implementing Collection interface, etc.), the tool leverages (interprocedural) def-use chains to transitively resolve the resource r_2 flowing to r_1.

If a data flow is observed between r_1 and r_2, Poirot adds the following bi-directional implication constraint (depicted in Rule R_{14}):

$$\Big(AccessControl(p, r_1, d) \xrightarrow{0.80} AccessControl(p, r_2, \texttt{-})\Big) \land \Big(AccessControl(p, r_2, d) \xrightarrow{0.80} AccessControl(p, r_1, \texttt{-})\Big)$$

Parameter Flow

We observe a special type of data flow constraint that can help us refine the less certain 1-n reachability constraints. A parameter flow from an API resource r_1 to a reachable resource r_2 often hints that r_2 is highly related to r_1. We employ this observation to refine the protection probabilities of reachable resources.

The confidence is calculated as a function of the number of parameters that flow to a target resource. If a high-confidence parameter flow is observed between an API resource r_1 and a reachable resource r_2, Poirot adds the following implication constraint (depicted in Rules R_{15} and R_{16}):

$$AccessControl(p, a, d) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$$

$$AccessControl(p, a, \texttt{BACKWARD}) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$$

5.6.4 Access Control Aggregation.

At this stage, Poirot has gathered a set of access control and implication constraints, each denoting our confidence that a resource r requires a protection p. We note that these confidences are obtained via different directions (e.g., `FORWARD`, `SELF`, etc.). We enable the inference engine to aggregate the confidence into a final confidence via Rule R_{17} in Table 5.2. Specifically, given a propagation direction d where d is not `AGGREGATED`, the confidence of $AccessControl(p, r, d)$ is faithfully propagated to $AccessControl(p, r,$ `AGGREGATED`$)$. If a protection recommendation is derived from different directions, the aggregated confidence will subsequently increase. The aggregated confidence also increases as new facts and observations of the same type are recovered at multiple program points.

5.7 Poirot in Action

We implement a prototype for Poirot consisting of two components: (1) a static analysis component and (2) a probabilistic inference engine. The static analysis component is built on top of WALA [20] and relies on Akka Typed [13] to parallelize the analysis. The probabilistic inference engine is built on ProbLog [19], a state-of-the-art probabilistic inference engine. As the underpinning solving technique is beyond the scope of this paper, we omit the details.

The static analyzer processes the Android framework, extracts basic facts and accordingly generates access control constraints. The analyzer implements a number of *Observation Extraction* modules, each responsible for identifying structural, semantic or data-flow observations. The analyzer further generates corresponding implication constraints in the form of Probabilistic Logic Program rules – i.e., $C \wedge x_1 \xrightarrow{p} x_2$.

The constraint solver associates each `Resource` r with one or more `Recommendations`. Each `Recommendation` consists of a `Protection` r_p and a `Confidence` c, where c is a value between 0 and 1. The tool outputs a ranked list of recommendations, from which we pick the top three results (Refer to Section 5.8.2). We normalize the recommendations following [21] to allow comparison and effective inconsistency detection.

Example. We use the AOSP API `getSyncStatusAsUser(...)` defined in the `ContentService` to illustrate Poirot's output. The tool generates three protections recommendations with the following probabilities:

1. `android.permission.INTERACT_ACROSS_USERS` $\wedge$ `android.permission.READ_SYNC_SETTINGS` with probability 0.91.

2. `android.permission.INTERACT_ACROSS_USERS_FULL` ∧ `android.permission.READ_SYNC_SETTINGS` with probability 0.91.

3. `android.permission.INTERACT_ACROSS_USERS_FULL` ∧ `android.permission.READ_SYNC_STATS` with probability 0.91.

Observe that the above recommendations are disjunctive, meaning that just one is sufficient for proper access control enforcement. To detect inconsistencies, Poirot compares the recommended access control enforcement with the implemented access control after normalization. Since the API implements the third recommendation, the case is considered consistent.

5.8 Evaluation

We design several experiments that assess Poirot's effectiveness and performance. Specifically, our evaluation aims to answer the following research questions:

- **RQ1:** Can Poirot accurately infer protection recommendations for Android resources?
- **RQ2:** Can different cut-off criteria configurations affect Poirot's accuracy?
- **RQ3:** Can variations in the probability values affect Poirot's accuracy?
- **RQ4:** What is the impact of each probabilistic rule on the analysis results?
- **RQ5:** What is Poirot's runtime and memory overhead?
- **RQ6:** Can Poirot accurately detect access control inconsistencies?
- **RQ7:** Can Poirot detect a greater number of access control inconsistencies than state-of-the-art tools?
- **RQ8:** Can Poirot suppress the false alarms associated with state-of-the art inconsistency detection tools?

All experiments were conducted on an IBM Power LC922 server machine equipped with a 22 core CPU (2.6 GHz POWER9 processor) and 256G main memory.

5.8.1 (RQ1) Evaluating Poirot's Protection Recommendations

In this experiment, we evaluate the accuracy of Poirot's protection recommendations for framework APIs.

Computation of Accuracy

Before describing our experiment setup, we explain how we estimate the accuracy of Poirot's generated protection recommendations. For each API, Poirot outputs a ranked list of protection recommendations with probabilities. Intuitively, when the calculated probability of a recommendation is *sufficiently high*, we can conclude that the API does indeed require the recommended protection. We introduce a configurable parameter `CUTOFF` and only report the protection recommendations with probabilities higher than `CUTOFF`. Note that more than one recommendation may correctly satisfy the latter condition due to the disjoint nature of Android access control. Thus, we introduce another configurable threshold $\texttt{TOP}_n$ to limit the number of reported recommendations. $\texttt{TOP}_n$ denotes the optimum number of protections that Poirot should report. We consider a recommendation for an API to be *accurate* if at least one recommended access control in the $\texttt{TOP}_n$ recommendations is as strong as the enforced access control found within the implementation of the API in AOSP, which we rely on as ground truth. Unless otherwise specified, we report the accuracy based on the configurations $CUTOFF$=90% and TOP_n=3. (Refer to Section 5.8.2 for more details on the selection criteria.)

Experiment Setup

For each AOSP system service, we begin by gathering all service APIs. We randomly select 10% of these APIs, which we term the *testing set*. Our goal is to generate accurate, high-confidence recommendations for the testing set APIs using basic facts generated from the other 90% of APIs, which we term the *training set*. We repeat this process ten times so that all service APIs are part of the testing set at least once.

Each round, we gather basic facts *only from the training APIs*. We supplement the basic facts with implication constraints from APIs in either set. Then, we pass all basic facts and constraints into the inference engine and attempt to output high-confidence recommendations for the testing set APIs. Finally, we compare all high-confidence recommendations with the corresponding AOSP API implementations to assess the recommendation accuracy.

We rely on two additional setups to assess the impact of increasing the pool of APIs used to derive the training and testing sets. The first additional setup considers APIs from two similar-in-name services at one time. The second additional setup considers three similar-in-name services at one time.

Table 5.3: Evaluation of APIs with High Confidence Access Control Recommendations

		No Unlinked Resources			Unlinked Resources >=1			
Set	Avg. APIs Analyzed	APIs (#)	Total Satisfaction	Partial Satisfaction	APIs (#)	Total Satisfaction	Partial Satisfaction	Correct Recommendations
1-system	78	59	56	1	19	3	1	77%
2-system	131	101	101	0	30	3	7	82%
3-system	175	136	129	3	39	6	10	84%

Results

Table 5.3 reports the results. Column 1 lists the evaluation sets that we used for training and Column 2 reports the average number of APIs for which Poirot was able to generate a high confidence protection recommendation. As expected, the number of APIs for which Poirot produces a recommendation increases as we include more services in the analysis.

Our analysis distinguishes between APIs with *linked resources* and those with *unlinked resources*. A linked resource is a sink within a testing API that is associated with a high-confidence recommendation. Recommendations for a linked resource can be propagated back up to the testing API. On the other hand, an API with unlinked resources contains sinks with no corresponding high-confidence recommendations. As a result, an inaccurate recommendation in a testing API with an unlinked resource could be attributed to the fact we did not extract basic facts from some related APIs also in the testing set.

Columns 3-8 report the number of APIs for which Poirot generated *a high confidence recommendation*. Overall, Poirot achieves an accuracy of 77%, 82%, and 84%, in 1-system, 2-system, and 3-system service sets. As expected, the accuracy improves as more services are included in the analysis, leading Poirot to uncover new cross-service observations and thus sharpen in-service probabilities.

5.8.2 (RQ2) Impact of Cut-off Criteria

This experiment evaluates the impact of the $CUTOFF$ and TOP_n criteria. Columns 3-6 in Table 5.4 report Poirot's accuracy using four TOP_n settings (namely, 1, 2, 3, and 4) and under three $CUTOFF$ configurations (0.85, 0.9, and 0.95). The last column reports the coverage achieved. Note that the impact of TOP_n on the coverage is negligible; hence, we report the coverage based on the $CUTOFF$ criteria only. As shown, Poirot achieves the highest accuracy at $CUTOFF = 0.95$ and at $TOP_n = 3$ or TOP_n 4 – there is no significant improvement at top 4 for all $CUTOFF$ configurations. Observe that $CUTOFF$ impacts the coverage in the other direction. This experimentation demonstrates that $CUTOFF = 0.9$ and $TOP_n = 3$ leads to an optimal trade-off between accuracy and coverage.

Table 5.4: Impact of Cut-off criteria.

		Accuracy (%)				**Coverage (%)**
		TOP 1	TOP 2	TOP 3	TOP 4	
	0.85	74.3	74.6	75.2	75.3	60.2
CUTOFF	0.9	76.6	76.7	77.4	77.4	59.4
	0.95	78.9	81.4	82.7	82.7	55.6

5.8.3 (RQ3) Impact of Prior Probability Values

We examine the sensitivity of Poirot's accuracy to variations in the constraints' prior probability values. We run the analysis under multiple configurations for two representative constraints: (1) the *Getter-to-Setter* constraint with confidence varying from 0.8 to 0.9 and (2) the *Reachability* constraint with confidence varying from 0.50 to 0.60. As shown in Table 5.5, the exploration demonstrates that parameter variation does not significantly affect the results as the accuracy varies within a limited range of less than 2%. Note that variations in other constraints, which we omit due to space limits, reveal similar trends. This experiment shows that Poirot is robust against prior probability variations.

Table 5.5: Accuracy (%) of Poirot under different prior probabilities for two constraints.

		Getter-to-Setter Constraint		
		p = 0.8	p = 0.85	p = 0.9
Reachability	p = 0.5	77.61	77.72	77.88
	p = 0.55	78.57	77.46	76.99
Constraint	p = 0.6	77.98	78.1	77.31

5.8.4 (RQ4) Impact of Probabilistic Constraints

In this experiment, we estimate the impact of Poirot's collected constraints on the probabilistic inference. Each constraint's impact can be understood by examining its frequency,

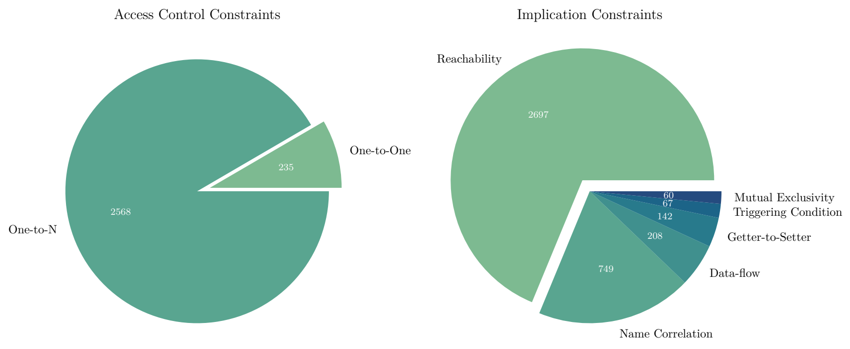


Figure 5.5: Breakdown of Probabilistic Constraints in AOSP

as the number of collected constraints plays a major role in the inference. To conduct this experiment, we rely on a similar setup to Experiment 5.8.3 on AOSP. We count and report the number of each constraint type found and present them in Figure 5.5. In total, Poirot collects 2803 access control constraints and 3923 implication constraints from AOSP. Though all constraints contribute to the inference, reachability and 1-n constraints are particularly prevalent.

5.8.5 (RQ5) Runtime and Memory Overhead

Table 9 shows the execution time and memory consumption of Poirot on the analyzed ROMs. The results are broken down per analysis phase. Poirot's main bottleneck is the basic facts extraction process, which relies on a path-sensitive, inter-procedural analysis. The execution time varies for different ROMs. For highly customized images, such as the Amazon Fire HD, the analysis takes more time.

Table 5.6: Average Overhead Measurement

ROM	**Basic Fact Extraction**		**Implication Constraint Generation**		**Probabilistic Inference**		**Inconsistency Analysis**	
	Time (min)	Memory (mB)	Time (min)	Memory (mB)	Time (min)	Memory (mB)	Time (min)	Memory (mB)
AOSP	50.05	332.29	22.80	302.08	23.50	367.49	2.70	365.44
Xiaomi Poco C3	53.31	373.06	33.35	280.83	30.30	361.48	4.05	366.27
Amazon Fire HD	56.00	301.89	32.42	309.26	30.98	320.25	4.47	382.29
LG LM-V405	54.13	327.10	31.35	300.85	28.02	338.24	3.34	367.42

5.8.6 (RQ6 & RQ7) Detecting Inconsistencies

This experiment evaluates Poirot's ability to detect access control inconsistencies. We analyze four ROMs from AOSP, Amazon, Xiaomi, and LG. Detailed information about the ROMs is listed in Columns 1 and 2 in Table 5.7.

Experiment Setup

Unlike Experiment 5.8.1, we extract basic facts from *all APIs* since a diverse set of basic facts is necessary to accurately detect access control inconsistencies. We pass these basic facts and all generated implication constraints to Poirot's inference engine in order to generate high-confidence recommendations that can be used to detect inconsistencies. An *inconsistency* is reported when Poirot's high-confidence protection recommendation for an API is *stronger* than the API's enforced access control.

Table 5.7: Inconsistency Detection Results of Poirot

Rom	**Version**	**Analyzed APIs**	**Inconsistencies (TP)**	**With >= 1 implication constraint**
AOSP	10	2739	8 (5)	1
Xiaomi Poco C3	10	3335	19 (14)	4
Amazon Fire HD	10	2779	18 (12)	4
LG LM-V405	10	1585	15 (10)	4*

*one case exposes and impacts 118 APIs

Results

Table 5.7 presents the reported inconsistencies. As shown, Poirot detects high-confidence true positive (TP) inconsistencies in all analyzed ROMs, ranging from 5 in AOSP to 14 in Xiaomi – in total, *26 unique inconsistencies.* It is worthy to note that one instance in LG [4] exposes 118 APIs, each leading to a different security impact including obtaining runtime permissions, starting apps with system privilege, and even enforcing a password recovery. Notwithstanding the high-severity level and tremendous amount of the exposed APIs, we consider the 118 cases as a single inconsistency.

[4]This case was illustrated in Figure 5.2.

Inconsistencies Uniquely Discovered by Poirot

Column 5 in Table 5.7 lists the number of inconsistencies that were detected using at least one non-reachability implication constraint. As shown, 10 inconsistencies were detected uniquely by Poirot. This means that our tool was able to *uniquely detect* 38% of all detected inconsistencies. We have manually analyzed the implementation of each reported inconsistency to estimate this number.

Poirot's False Positives

Due to the lack of ground truth security specifications for custom vendor APIs, we estimate the false positive (FP) inconsistencies through manual investigation. We report the number of FPs in column 4. As shown, out of all reported inconsistencies, 32.7% are false alarms. We identified two main reasons for the false positives. First, certain high-confidence recommendations were derived from *substantially frequent occurrences* of low-confidence constraints. In such cases, the higher number of constraints improves the initially assigned low protection probabilities. Second, our tool failed to recognize some *custom* access control checks uniquely introduced by vendors.

5.8.7 (RQ8) Suppressing False Positives of Other Tools

This experiment assesses whether Poirot successfully suppresses the high false positives seen in Kratos [70] and AceDroid [21], two state-of-the-art access-control inconsistency detection tools. Both tool operate in a largely similar fashion with subtle differences. To detect inconsistencies, Kratos performs a simplistic convergence analysis, while AceDroid relies on access control modeling and normalization to detect exploitable inconsistencies only.

We obtained access to AceDroid and applied it to analyze the collected ROMs. Since Kratos is not publicly available, we developed a simulated version, which we refer to as Kratos+. Kratos relies on a number of unknown heuristics to reduce the number sinks used to find converging APIs. To ensure a faithful comparison with Poirot, we incorporate Poirot's sink reduction strategy into Kratos.

Experiment Setup

We applied AceDroid and Kratos+ to identify inconsistencies. We estimate FPs using the notion of *likely protection targets*, which we explain next. A *protection target* is a sink within an API that is the target of some access control enforcement. A *likely protection target* is a sink that we believe has strong potential to be a *protection target* because Poirot identified it as linked to the calling API through some implicit relation, such as a naming correlation or a parameter flow. Intuitively, if AceDroid or Kratos+ detect an inconsistency for two APIs that converge upon an *unlikely protection target*, then that inconsistency is probably an FP.

Table 5.8: False Positives of AceDroid and Kratos+.

ROM	**AceDroid**			**Kratos+**		
	Inc#	FP# (%)	FP (%) ↓ by Poirot	Inc#	FP# (%)	FP (%) ↓ by Poirot
AOSP	27	22 (81.4)	54.5	51	46 (90.1)	58.9
Xiaomi Poco C3	44	34 (77.2)	66.3	88	78 (88.6)	70.6
Amazon Fire HD	34	26 (76.4)	56.8	86	79 (91.8)	64
LG LM-V405	39	28 (71.9)	54.1	73	64 (87.6)	62.3

Results

Table 5.8 reports the results. As shown, both AceDroid and Kratos+ generate substantial FPs ranging from 71% to 81% in AceDroid, and from 85% to 91% in Kratos+. We note that both estimations are higher than the FPs reported by AceDroid and Kratos. We believe this is likely due to the fact that we are not including the heuristics and manual filtering followed by AceDroid and Kratos to reduce the number of sinks. Although our results are an over-estimation of the existing work's FPs, we note that they reflect pure-convergence inconsistency detection results.

False Positive Suppression by Poirot

As shown in Columns 4 and 7 in Table 5.8, Poirot substantially improves the results of Kratos and AceDroid thanks to its ability to pinpoint likely protection targets in APIs. It can reduce the false positives up to 66% and 70% in AceDroid and Kratos, respectively.

5.9 Case Study

We would like to note that not all inconsistencies are exploitable. The reasons are twofold. First, triggering an inconsistency may require certain conditions unrelated to access control to be met. These are not picked up by our tool. Second, an API's functionality might not necessarily reflect a security sensitive operation.

Table 5.9 reports the cases for which we have successfully built a PoC. Next, we select one compelling case for discussion. We intentionally picked a vulnerability that is hard to detect using existent inconsistency detection tools.

Table 5.9: Summary of Discovered Protection Inconsistencies that can lead to Security Issues

OS Image	System Service:API	Enforced Access Control	Recommended Access Control	Constraint(s)	Potential Security Implication	Report Status
LG LM-V405	LGMDM.setActiveAdmin	UserCheck AND (E —— MANAGE_DEVICE_ADMINS)	UserCheck AND (SYSTEM_PERMISSION)	Trigger Condition Reachability	Replace device admin with own package Expose 118 APIs in MDM service	Ack / Fixed
LG LM-V405	LGMDM. getRunningPackagesFromPid	UserCheck	UserCheck AND (REAL_GET_TASKS)	Data Flow Reachability	Exfiltrate running packages details	Ack / Fixed
Fire HD 10	AmazonInput.setInputFilter	E	SYSTEM_PERMISSION	Reachability Naming Correlation	Key Logger	Ack Planned patch
Fire HD 10	MigrationService.migrate	Normal_Permission	MOVE_PACKAGE	Setter-getter Naming Correlation Forward Reachability	Local System crash Reboot	Ack Planned patch
Fire HD 10	MigrationService.getMigrateData	Normal_Permission	MOVE_PACKAGE	Data Flow	Obtain migration meta data	Under Analysis
Fire HD 10	AmazonPMS.setAmazonFlags	E	SYSTEM_PERMISSION	Trigger Condition	Change Amazon-Specific Package Settings	Under Analysis
Fire HD 10	AmazonPMS.removeAmazonFlags	E	SYSTEM_PERMISSION	Trigger Condition	Change Amazon-Specific Package Settings	Reported
Xiaomi Poco C3	IPerfShielder. getAllRunningProcessMemInfos	E	UserCheck AND (REAL_GET_TASKS)	Reachability	Exfiltrate running processes Info	Ack*

*Xiaomi has acknowledged the issue but mentioned it was reported by a different party before us.

Crashing and Rebooting the System

Poirot reported two inconsistencies in Amazon Fire HD's MigrationService, located in two custom APIs. While both APIs do enforce a `Normal` permission, our tool recommended a higher privilege check: a permission equivalent to the system-level permission `MOVE_PACKAGE`. We manually investigated the reports and found that Poirot generated a few high confidence recommendations for different resources within the two APIs based on a combination of data-flow, backward reachability and naming correlation hints. The detection entailed a cascading effect that propagated a protection from a single occurrence of a basic access control fact to two privileged resources. Specifically:

- Poirot identified a data-flow hint that assigned a global field the return value of a privileged getter API with assigned protection `MOVE_PACKAGE`.
- Poirot relied on the data-flow hint to propagate protection `MOVE_PACKAGE` to the global field; implying that any corresponding read operation should require this protection.
- Poirot identified an API `getMoveData` that reads and returns global field; as such, the `MOVE_PACKAGE` recommendation was issued for the `getMoveData` API. The case was

flagged as an inconsistency since `getMoveData`'s enforced access control was weaker than `MOVE_PACKAGE`.

- In a different API, Poirot identified a getter-to-setter hint, where the global field was being written. Hence, Poirot concluded that the new site requires `MOVE_PACKAGE`.
- The recommendation was further consolidated by naming correlation and backward reachability hints pertaining to another resource. Details are elided for simplicity. The API `migrate` was subsequently flagged as an inconsistency due to a weaker protection enforcement.

We have tested the reported vulnerability and found that the two APIs indeed lack protections. Concerningly, triggering `migrate` with specific parameters (i.e., supplying private data folder to be migrated) crashes the system server.

Chapter 6

Conclusion

This thesis presents two novel approaches to detect access control flaws in the Android framework.

We introduce ReM, a bloated custom Residual API detector, and perform the first large-scale, longitudinal study on the security impacts of customization-induced code bloat in the Android framework. We find that all major Original Equipment Manufacturer (OEM) code bases contain these unused remnant APIs. Using ReM's suite of static analysis techniques to detect Residuals and analyze their security flaws, we discover that Residuals open the door to serious security vulnerabilities, including access control inconsistencies and undefined security attributes.

We also present our tool Poirot, which advances the state-of-the-art inconsistency detection approaches by considering the inherent imprecision of the linkage between a resource and a protection. The tool goes beyond a simplistic reachability analysis to also incorporate insights from structural, semantic and data-flow relations between resources and protections. Poirot relies on static analysis techniques to extract implicit relations between resources and protections. The tool then leverages probabilistic inference to make sense of those relations and output final recommendations. Our evaluation of Poirot finds that its probabilistic approach does indeed reduce the false positives experienced by existing inconsistency detection tools. Furthermore, Poirot detects new implicit inconsistencies overlooked by existing inconsistency detection approaches.

References

[1] Permissions on Android. https://developer.android.com/guide/topics/permissions/overview, 11 2020.

[2] Platform Architecture. https://developer.android.com/guide/platform, 5 2020.

[3] The Heartbleed Bug. https://heartbleed.com, 2020.

[4] apktool. https://ibotpeaches.github.io/, 2021.

[5] baksmali. https://github.com/JesusFreke/smali, 2021.

[6] Dynamic partitions. https://source.android.com/devices/tech/ota/dynamic_partition, 2021.

[7] imjtool. http://newandroidbook.com/tools/imjtool.html, 2021.

[8] lpunpack. https://github.com/LonelyFool/lpunpack_and_lpmake, 2021.

[9] oat2dex. https://github.com/testwhat/SmaliEx, 2021.

[10] Salt. https://github.com/steadfasterX/SALT, 2021.

[11] simg2img. https://github.com/anestisb/android-simg2img, 2021.

[12] vdexexctractor. https://github.com/anestisb/vdexExtractor, 2021.

[13] Akka: Build powerful reactive, concurrent, and distributed applications more easily. https://akka.io/, 2022.

[14] Ccve-2022-20204. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-20204, 2022.

[15] Cve-2022-20126. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-20126, 2022.

[16] Cve-2022-20192. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-20192, 2022.

[17] Cve-2022-20206. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-20206, 2022.

[18] FReD: Identifying file Re-Delegation in android system services. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[19] Problog. https://dtai.cs.kuleuven.be/problog/, 2022.

[20] Wala. https://github.com/wala/WALA, 2022.

[21] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. Internet Society, 2 2018.

[22] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise android API protection mapping derivation and reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1151–1164. Association for Computing Machinery, 10 2018.

[23] 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.

[24] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1248–1259, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting inconsistent security configurations in custom android ROMs via differential analysis. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1153–1168, Austin, TX, August 2016. USENIX Association.

[26] ACM. Acm ccs 2021. https://www.sigsac.org/ccs/CCS2021/, 2021.

[27] Open Handset Alliance. Faq. https://www.openhandsetalliance.com/oha_faq.html, 2007.

[28] Open Handset Alliance. Industry leaders announce open platform for mobile devices. https://www.openhandsetalliance.com/press_110507.html#:~:text=MOUNTAIN%20VIEW%2C%20Calif.%3B%20BONN,comprehensive%20platform%20for%20mobile%20devices, 2007.

[29] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *CCS*, page 1070, 2012.

[30] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, Santa Clara, CA, August 2019. USENIX Association.

[31] 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.

[32] Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: System-centric IPC provenance on android. In *ACM International Conference Proceeding Series*, volume 2014-December, pages 36–45. Association for Computing Machinery, 12 2014.

[33] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 46–55, New York, NY, USA, 2014. Association for Computing Machinery.

[34] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Pasareanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 866–877, 2015.

[35] Michael D. Brown and Santosh Pande. Carve: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Michael D. Brown and Santosh Pande. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[37] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 810–821, 2016.

[38] Bonnie Cha and Nicole Lee. Review: Google's htc dream phone – that's it? http://www.cnn.com/2008/TECH/ptech/10/27/cnet.tmobile.g1/index.html, 2009.

[39] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1253–1270, Vancouver, BC, August 2017. USENIX Association.

[40] Domenico Cotroneo, Antonio Ken Iannillo, and Roberto Natella. Evolutionary fuzzing of android OS vendor system services. *CoRR*, abs/1906.00621, 2019.

[41] Abdallah Dawoud and Sven Bugiel. Bringing balance to the force: Dynamic analysis of the android application framework. *Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework*, 2021.

[42] Alastair F. Donaldson, Alice Miller, and David Parker. Language-level symmetry reduction for probabilistic model checking. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, pages 289–298, 2009.

[43] Joshua Drake. Stagefright: Scary code in the heart of android, 2015.

[44] 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.

[45] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[46] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. page 726. ACM, 2011.

[47] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 341–350, 2011.

[48] Roberto Gallo, Patricia Hongo, Ricardo Dahab, Luiz C. Navarro, Henrique Kawakami, Kaio Galvão, Glauber Junqueira, and Luander Ribeiro. Security and system architecture: Comparison of android customizations. WiSec '15, New York, NY, USA, 2015. Association for Computing Machinery.

[49] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 166–176, 2012.

[50] 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. 1 2019.

[51] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 156–164, 2004.

[52] Roee Hay. fastboot oem vuln: Android bootloader vulnerabilities in vendor customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[53] Grant Hernandez, Swarnim Yadav, Byron J Williams, Kevin RB Butler, Dave Tian, Anurag Swarnim Yadav, and Kevin R B Butler. BigMAC: Fine-Grained Policy Analysis of Android Firmware BIGMAC: Fine-Grained Policy Analysis of Android Firmware. In *Proceedings of the 29th UNSENIX Security Symposium*, 2020.

[54] Qinsheng Hou, Wenrui Diao1, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guol, Yuanzhi Li, Meining Nie, and Haixin Duan.

[55] 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, Washington, D.C., August 2015. USENIX Association.

[56] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A gray-box android fuzzer for vendor service customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11, 2017.

[57] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–199, 2018.

[58] Kailani R. Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G. Bardas. *Deploying Android Security Updates: An Extensive Study Involving Manufacturers, Carriers, and End Users*, page 551–567. Association for Computing Machinery, New York, NY, USA, 2020.

[59] Renuka Kumar, Sreesh Kishore, Hao Lu, and Atul Prakash. Security analysis of unified payments interface and payment apps in india. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1499–1516. USENIX Association, August 2020.

[60] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 585–591, 2011.

[61] Clement Lecigne and Christian Resell. Protecting android users from 0-day attacks, 2022.

[62] Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 254–264, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Lookout. Pegasus for android technical analysis and findings of chrysaor, 2017.

[64] Kenneth A. Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software*

Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pages 1187–1198. IEEE / ACM, 2019.

[65] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. Binary debloating for security via demand driven loading, 2019.

[66] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, but verify: A longitudinal analysis of android oem compliance and customization. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 87–102, 2021.

[67] 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&amp;P 2021, 42nd IEEE Symposium on Security and Privacy, 23-27 May 2021, Virtual Conference*, 2021.

[68] Jan Ruge. Cve-2020-0022 an android 8.0-9.0 bluetooth zero-click rce – bluefrag, 2020.

[69] Michael Schwarz, Moritz Lipp, and Daniel Gruss. Javascript zero: Real javascript and zero side-channel attacks. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[70] 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.

[71] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security, 2017.

[72] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[73] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. A security policy oracle: Detecting security holes using multiple api implementations. *ACM SIGPLAN Notices*, 46(6):343–354, 2011.

[74] StatCounter. Mobile operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2022.

[75] 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.

[76] Neil Toronto, Jay McCarthy, and David Van Horn. Running probabilistic programs backwards. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 53–79, 2015.

[77] Guliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A. Gunter. Resolving the Predicament of Android Custom Permissions. Internet Society, 2 2018.

[78] Jeffrey A Vaughan and Stephen Chong. Inference of expressive declassification policies. In *2011 IEEE Symposium on Security and Privacy*, pages 180–195. IEEE, 2011.

[79] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. {JIGSAW}: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 973–988, 2014.

[80] Wikipedia contributors. Sørensen–dice coefficient — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=z%C3%B8rensen%E2%80%93Dice_coefficient&oldid=1083624728, 2022. [Online; accessed 14-June-2022].

[81] 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.

[82] Hao Xia, Yuan Zhang, Yingtian Zhou, Xiaoting Chen, Yang Wang, Xiangyu Zhang, Shuaishuai Cui, Geng Hong, Xiaohan Zhang, Min Yang, and Zhemin Yang. How android developers handle evolution-induced api compatibility issues: A large-scale study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 886–898, New York, NY, USA, 2020. Association for Computing Machinery.

[83] 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.

[84] 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.

[85] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Hui Liu, Qing Wang, Yueheng Zhang, and Dawu Gu. Show me the money! finding flawed implementations of third-party in-app payment in android apps. In *NDSS*. The Internet Society, 2017.

[86] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. Netplier: Probabilistic network protocol reverse engineering from message traces. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[87] 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. In *Proceedings of the Web Conference 2021*, WWW '21, page 2733–2744, New York, NY, USA, 2021. Association for Computing Machinery.

[88] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1663–1674, New York, NY, USA, 2016. Association for Computing Machinery.

[89] 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.

[90] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. OSPREY: recovery of variable and data structure via probabilistic analysis for stripped binary. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 813–832. IEEE, 2021.

[91] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[92] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, pages 91–100, 2008.

[93] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android.

[94] 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.