Poirot: Probabilistically Recommending Protections for the Android Framework

Zeinab El-Rewini
University of Waterloo
zelrewin@uwaterloo.ca

Zhuo Zhang
Purdue University
zhan3299@purdue.edu

Yousra Aafer
University of Waterloo
yousra.aafer@uwaterloo.ca

ABSTRACT

Inconsistent security policy enforcement within the Android framework can allow malicious actors to improperly access sensitive resources. A number of prominent inconsistency detection approaches have been proposed in and across various layers of the Android operating system. However, the existing approaches suffer from high false positive rates as they rely solely on simplistic convergence analysis and reachability based relations to reason about the validity of access control enforcement.

We observe that resource-to-access control associations are highly uncertain in the context of Android. Thus, we introduce Poirot, a next-generation inconsistency detection tool that leverages probabilistic inference to generate a comprehensive set of protection recommendations for Android framework APIs. We evaluate Poirot on four Android images and detect 26 total inconsistencies.

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**; **Access control**; • **Mathematics of computing** → *Probabilistic inference problems.*

KEYWORDS

android security; inconsistency detection; probabilistic inference

ACM Reference Format:
Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. 2022. Poirot: Probabilistically Recommending Protections for the Android Framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560710

1 INTRODUCTION

Access control systems are known to be vulnerable to anomalies in security policies, such as inconsistent security policy enforcement. The Android security model is no exception. Prominent research efforts [6, 18, 22] have shown that the Android framework, which houses the framework system services and implements the Application Programming Interfaces (APIs), is riddled with access control inconsistencies. An inconsistency occurs when one path to a sensitive resource (which can take the form of a field access, internal method or API invocation) 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.

Prior works provide good approximate solutions to detecting framework-level access control inconsistencies. Kratos [22], AceDroid [6] and ACMiner [18] all rely on *convergence analysis* to assess the adequacy of enforced access control. The tools extract access control enforcement along different paths leading to a *reachable shared convergence point* and compare them to detect inconsistencies. AceDroid advances Kratos's approach by modeling and normalizing access control checks to reduce false alarms. New approaches attempt to detect framework-level access control inconsistencies by leveraging security specifications across different Android software stack layers. FReD [2] identifies conflicting access control requirements for APIs by comparing them against their reachable files' Linux-layer permissions. IAceFinder [34] compares access control enforcement in both the Java and native contexts to detect cross-context inconsistencies.

While this body of literature has proven to be quite beneficial, we note that the existing works suffer from a number of shortcomings. First, cross-layer inconsistency detection solutions are limited in scope as they can only detect vulnerabilities in APIs with specific implementations (i.e., APIs accessing files as in FReD [2] or APIs reaching a JNI interface as in IAceFinder [34]). Second, although in-framework inconsistency detection approaches leverage a richer learning ground for access control owing to the substantial number 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, 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. Additionally, the tools rely only on a *reachability* analysis to link resources and derive their access control. However, we observe that Android resources are also connected via implicit structural, semantic and data-flow relations. 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 may indicate that the resource is likely to require the API's protection. Modeling these implicit relations can uncover new inconsistencies.

The fundamental limitation of existing tools is that *statically determining which protections are required for certain resources is highly imprecise.* On the one hand, in a given Android API, a protection check may precede both security-relevant and non-security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '22, November 7–11, 2022, Los Angeles, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560710

relevant resources. On the other hand, deducing an access control implication from an implicit relation linking resources (such as a naming similarity) entails a degree of uncertainty.

In this work, we re-conceptualize the inconsistency detection problem to account for uncertainty via probabilistic inference. Instead of assuming precise associations between resources and access control (resource r requires protection p), we assume probabilistic ones (resource r may require protection 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 a prior probability value indicating our degree of belief in the access control implication. We then propagate the initially assigned protections to other resources via *implication constraints*. These constraints encode *statically observed* structural, semantic and data-flow relations that connect resources and enable the propagation of their protections. To account for inherent uncertainty, this propagation is probabilistic.

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 discover 26 true inconsistencies. While some of these inconsistencies may be detected via existing approaches, we note that 10 are uniquely discovered by Poirot. We build end-to-end PoCs for 8 inconsistencies to demonstrate their security impacts. In particular, we note that one instance of an implicit relation has caused the exposure of 118 APIs, leading to substantial security impacts (e.g., acquiring permissions at runtime, enforcing a recovery password and others). We have responsibly reported the violations to the vendors; All vulnerabilities have been acknowledged and fixed.[1]

Our contributions are summarized as follows:

- We develop *Poirot*, a tool that generates probabilistic protection recommendations for framework-level resources. Poirot melds probabilistic inference and static program analysis to account for the uncertainties pertaining to static access control inference.
- Our proposed approach supplements the traditional reachability analysis with rich semantic, structural and data-flow relations that provide insight into the relationships between framework resources and protections.
- We evaluate Poirot on four Android images and find that it *substantially* suppresses false positives exhibited by AceDroid and Kratos, the two leading inconsistency detection tools, by 66% and 70%, respectively.

[1] Two vulnerabilities were internally known to the vendors and have been patched in newest models.

2 BACKGROUND AND MOTIVATION

In this section, we cover essential background and a few examples to motivate our proposed probabilistic inference approach.

2.1 Background

The Android framework is a collection of Java libraries and system services located within the Android middleware. Application developers rely on the framework APIs, the publicly exposed methods of the framework system services, to access integral Android features such as the camera, display settings or Bluetooth.

Each API implements a concise functionality by accessing one or many Android resources. These resources can largely be classified into three categories according to the taxonomy proposed in [9][2]: field access and update, internal method invocation (e.g., native methods, file access methods, non-exposed service methods) and API method invocation (an API may invoke another API).

Framework developers are responsible for implementing access control enforcement, depending on the category / sensitivity of accessed resources. For instance, the API `requestLocationUpdates()` in the `LocationManagerService` should require a location-access permission. An access control check determines whether (1) the calling app holds specific permissions or satisfies certain criteria (e.g., assigned a specific UID), and/or if (2) the calling physical user is privileged enough to access the resource.

Unfortunately, due to the lack of precise and complete security specifications, access control implementations tend to be uncoordinated and inconsistent. This has motivated the emergence of inconsistency detection solutions.

2.2 Motivation

To detect and correct access control inconsistencies in the Android framework, the Android community has proposed a number of inconsistency detection tools, which largely work by extracting and comparing access control enforcement along various paths leading to the same resource. Kratos [22] performs a path-insensitive analysis to extract *explicit* security checks (such as permission checks or package name checks) along the path to a given Android resource. The tool then conducts a convergence analysis to identify paths converging to the same resource and compares the *union* of extracted checks to detect potential inconsistencies. AceDroid [6] addresses specific Android access control peculiarities; namely, that access control implementations tend to be conjoint and/or disjoint and may be syntactically diverse yet semantically equivalent. It is able to do so by conducting a path-sensitive analysis, modeling a wider variety of security checks and normalizing diverse access control checks. While Kratos and AceDroid manually define

[2] Unlike [9], we do not classify exception throwing as a resource for simplicity reasons.

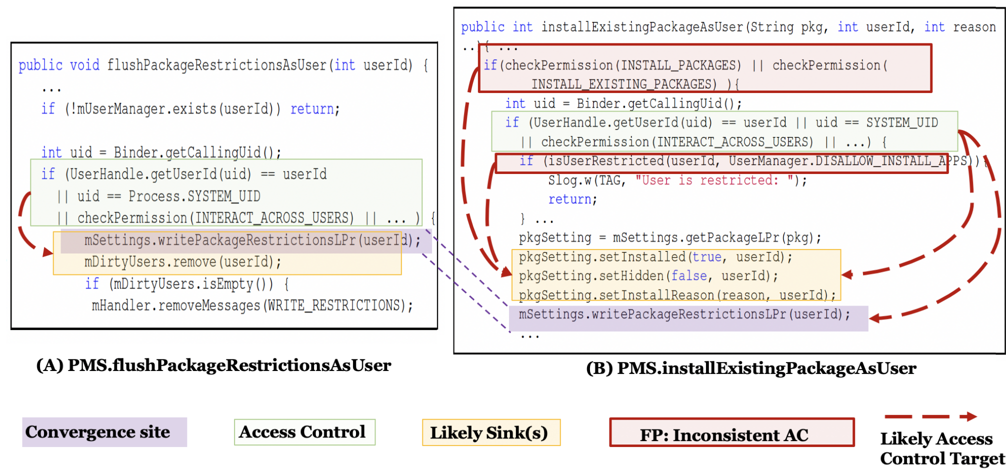

Figure 1: False Positive Due to Inaccurate Identification of Targets

patterns characterizing security checks, ACMiner [18] takes a semi-automated approach to security check identification by tracing back from thrown security exceptions.

Though the existing inconsistency detection tools have helped identify and correct significant access control anomalies, they suffer from two major limitations: (1) they may not accurately identify the targets of a given access control check and thus will inherently generate *an overwhelming number of false positives* and (2) they can only detect explicit reachability-based inconsistencies and thus may miss a significant number of other implicit inconsistencies. Next, we use examples to illustrate these shortcomings and explain how our technique addresses them.

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 a convergence exists, the tools 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 the most stringent access control path. However, this assumption is fundamentally inaccurate as 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 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 (`INSTALL_PACKAGES` or `INSTALL_EXISTING_PACKAGES`, shown in red) in addition to the user ownership/ privilege checks. 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. Existing tools would *wrongly* flag the least protected path leading to the convergence point (in this case, the path starting at the entry of `flushPackageRestrictionsAsUser()`) as a potential inconsistency since it does not enforce the checks shown 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 green) within `flushPackageRestrictionsAsUser()` likely target all operations shown in the yellow box (including the convergence point `writePackageRestrictionsLPr()`) since their names and parameter values imply a connection to flushing/ writing restrictions based on the user parameter. Note that we are uncertain about `mHandler.removeMessages()`'s relevance to the user check.
- The permission checks `INSTALL_PACKAGES` and `INSTALL_EXISTING_PACKAGES` and the user restriction check `DISALLOW_INSTALL_APPS` (in red) within `installExistingPackageAsUser()` likely target the methods `PkgSettings.setInstalled` and `PkgSettings.setInstallReason` since both their names are related to package installation.
- The user checks within `installExistingPackageAsUser()` (in green) likely target 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

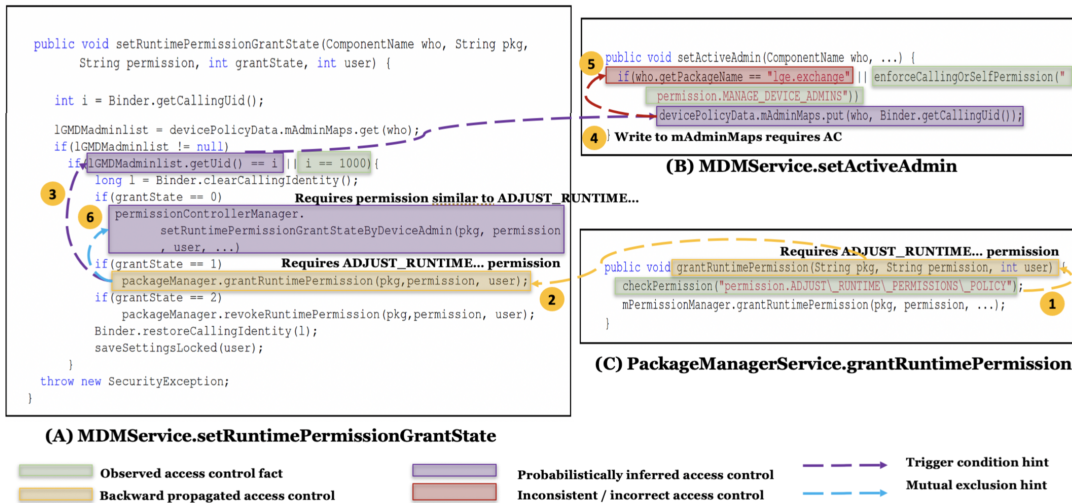

Figure 2: Probabilistic Inference of Access Control Checks and Implicit Inconsistencies

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 7.7).

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 1(A), the resources `mSettings.writePackageRestrictionsLPr()`, `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 [6]). 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, Android resources are usually transitively connected via multiple implicit relations. 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 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 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()` (in purple). Having full control of this important field allows the third-party app to subsequently trigger the underlying privileged operations (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 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 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 `setRuntimePermissionGrantState()`. This indicates the latter should enforce a permission with a minimum protection level equivalent to `ADJUST_RUNTIME_PERMISSION`. In (3), we observe that the API `setRuntimePermissionGrantState()` 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, `grantRuntimePermission()`. 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 discern another *hint* that helps 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.[3] As such, we can infer that both operations are *likely* to require similar access control. This

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

hint is particularly important when propagating the access control extracted from `grantRuntimePermission()` to LG's custom method `permissionControllerManager.setRun...Admin()`, 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.

Our Solution. Due to the *inherent uncertainty* in linking resources to protections, it is challenging to formulate general patterns that can precisely associate resources with access control. Hence, it is difficult to accurately detect access control anomalies. Statically extracted associations are imprecise for two main reasons: (1) accurately pinpointing the resource(s) targeted by an observed access control check is difficult and (2) inferring an access control implication from implicit observed associations entails a degree of uncertainty.

To meet these challenges, we propose a new solution centered around generating probabilistic protection recommendations for Android resources and leveraging them to identify 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. Specifically, depending on the type and number of hints collected, we compute the marginal probabilities that a resource r should be associated with various protections P. Observe that hints are inherently uncertain and may reflect different degrees of certainty. That is, some hints are more certain than others. The probabilistic analysis will aggregate these hints and their corresponding frequencies (i.e., a larger number of hints implies higher confidence) to suppress uncertainties and infer protection recommendations.

3 APPROACH

Given an Android ROM, Poirot preprocesses the framework and system classes to identify system services and their APIs. It statically analyzes the APIs to identify reachable 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 resources to be further analyzed.

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 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.

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; that is, 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 7.3.) 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.

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 Correlation, 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.

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

4 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.

4.1 Definitions

To facilitate discussion, we introduce a few Android-specific definitions in Figure 3. We use *func* to denote a `Function`, which can 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`), a `Protection` p (e.g., `Binder.getCallingUid() == 1000`) or others. We use s to denote a `Statement`, which represents a complete unit of execution. It corresponds to either a sequence of statements or code blocks along the true/false branches in conditional constructs.

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

Figure 3: A Simple Language for Android Functions

Table 1: Probabilistic Inference Rules

ID	Conditions*	Probabilistic Constraint
R_1	$ControlDependency(p, \{r\})$	$AccessControl(p, r, \texttt{SELF}) = true\ (0.95)$
R_2	$ControlDependency(p, R) \wedge \|R\| > 1 \wedge r \in R$	$AccessControl(p, r, \texttt{SELF}) = true\ (0.60)$
R_3	$Reachability(a, \{r\}) \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\}$	$AccessControl(p, a, d) \xrightarrow{0.95} AccessControl(p, r, \texttt{FORWARD})$
R_4	$Reachability(a, R) \wedge \|R\| > 1 \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\} \wedge r \in R$	$AccessControl(p, a, d) \xrightarrow{0.60} AccessControl(p, r, \texttt{FORWARD})$
R_5	$Reachability(a, R) \wedge r \in R \wedge d \in \{\texttt{BACKWARD}, \texttt{SELF}, \texttt{-}\}$	$AccessControl(p, r, d) \xrightarrow{0.95} AccessControl(p, a, \texttt{BACKWARD})$
R_6	$SameBlock(E(r_1), E(r_2))$	$AccessControl(p, r_1, \texttt{BACKWARD}) \xrightarrow{0.6} AccessControl(p, r_2, \texttt{FORWARD})$
R_7	$NameCorrelation(a, r) \wedge Reachability(a, \{r\}) \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\}$	$AccessControl(p, a, d) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$
R_8	$NameCorrelation(a, r) \wedge Reachability(a, \{r\}) \wedge InPath(p, a, r)$	$AccessControl(p, a, \texttt{BACKWARD}) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$
R_9	$NameCorrelation(a, r) \wedge Reachability(a, \{r\}) \wedge d \in \{\texttt{BACKWARD}, \texttt{SELF}, \texttt{-}\}$	$AccessControl(p, r, d) \xrightarrow{0.70} AccessControl(p, a, \texttt{BACKWARD})$
R_{10}	$\left(\exists s,\ s.t.\ s \equiv \texttt{if}\ (E(r_2^{read}))\ \{s_t\}\right) \wedge Contains(s_t, r_1) \wedge d \neq \texttt{AGGREGATED}$	$AccessControl(p, r_1, d) \xrightarrow{0.85} AccessControl(p, r_2^{write}, \texttt{-})$
R_{11}	$\left(\exists s,\ s.t.\ s \equiv \texttt{if}\ (e)\ \{s_t\}\ \texttt{else}\ \{s_f\}\right) \wedge (e \equiv E(\texttt{INPUT_CHK}) \vee e \equiv E(\texttt{SYS_PROPERTY})) \wedge Contains(s_t, E(r_1)) \wedge Contains(s_f, E(r_2)) \wedge d \neq \texttt{AGGREGATED} \wedge NameCorrelation(r_1, r_2)$	$(AccessControl(p, r_1, d) \xrightarrow{0.90} AccessControl(p, r_2, \texttt{-})) \wedge (AccessControl(p, r_2, d) \xrightarrow{0.90} AccessControl(p, r_1, \texttt{-}))$
R_{12}	$d \neq \texttt{AGGREGATED}$	$AccessControl(p, m^{getter}, d) \xrightarrow{0.80} AccessControl(p, m^{setter}, \texttt{-})$
R_{13}	$d \neq \texttt{AGGREGATED}$	$AccessControl(p, a^{getter}, d) \xrightarrow{0.80} AccessControl(p, a^{setter}, \texttt{-})$
R_{14}	$Data\text{-}flow(E(r_1), E(r_2))) \wedge d \neq \texttt{AGGREGATED}$	$(AccessControl(p, r_1, d) \xrightarrow{0.80} AccessControl(p, r_2, \texttt{-})) \wedge (AccessControl(p, r_2, d) \xrightarrow{0.80} AccessControl(p, r_1, \texttt{-}))$
R_{15}	$DataFlow(e, E(r)) \wedge Argument(a, e) \wedge Reachability(a, r) \wedge d \in \{\texttt{FORWARD}, \texttt{SELF}, \texttt{-}\}$	$AccessControl(p, a, d) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$
R_{16}	$DataFlow(e, E(r)) \wedge Argument(a, e) \wedge Reachability(a, r) \wedge InPath(p, a, r)$	$AccessControl(p, a, \texttt{BACKWARD}) \xrightarrow{0.70} AccessControl(p, r, \texttt{FORWARD})$
R_{17}	$d \neq \texttt{AGGREGATED}$	$AccessControl(p, r, d) \xrightarrow{1.00} AccessControl(p, r, \texttt{AGGREGATED})$

*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.

Our analysis considers three types of resources: (1) `FieldAccess`, denoted by f, (2) `InternalMethod`, denoted by m and (3) `API`, denoted by a. f is categorized by access type (read or write), while m and a are categorized as setters, getters or standard methods using a few rules and naming conventions.

Table 2: 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.

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

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

`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 4: Defining the Random Variables

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.

4.2 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 lead to the target resource, the analysis merges the in-path constraints for each ICFG path using a logical OR (implying disjoint checks). 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.

4.3 Access Control Constraints

Once the initial access control facts are collected, Poirot generates *access control constraints*, which assign prior probabilities to the random variables.

One-to-One Control-Dependency Constraints. One-to-one constraints link 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 1 shows that the permission check for MOUNT_UNMOUNT_FILESYSTEMS precedes a single call to mMoveCallbacks.unregister(). As such, we can intuitively link the permission to the invoke statement with high confidence.

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

Listing 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 1: $AccessControl(p, r, \mathsf{SELF}) = true$ (0.95) with the random variable $AccessControl(p, r, \mathsf{SELF})$ asserting that p is derived from a one-to-one control dependency. Figure 4 describes the meaning of *AccessControl*. Note that the third parameter denotes the propagation direction, which we will discuss shortly.

One-to-Many 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 1(A) and (B) for illustration.) Poirot formulates this access control constraint (depicted by R_2 in Table 1) for each pair of protection p and its control-dependent resources $r \in R$, as follows: $AccessControl(p, r, \mathsf{SELF}) = true$ (0.60) with the random variable $AccessControl(p, r, \mathsf{SELF})$ asserting that the protection p is derived from a one-to-many control dependency.

5 IMPLICATION CONSTRAINTS

Implication constraints do not directly link a resource with a protection. Instead, they link resources to one another by leveraging observed relations statically connecting the resources. As such, these constraints propagate protection recommendations across resources. Note that a propagated protection could be directly assigned by an 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 2 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.

5.1 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 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 $<API\text{-}r_{callee}>$ relations. Transitive reachability constraints will be encoded during probabilistic inference.

Observed reachability between *API* and r_{callee} implies that *API*'s inferred protections should be propagated to r_{callee}. However, we note that some of the inferred protections may already be encoded through control-dependency constraints. Consider Listing 2:

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

Listing 2: removeUser

The forward reachability hint between caller removeUser and callee removeUserUnchecked 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 4, Poirot considers five directions.

A SELF direction indicates that the protection is derived from a basic fact within the API's implementation. A FORWARD direction indicates that 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. In this case, removeUser's protection is propagated back to some other invoking API. A - direction signifies a direction-free propagation. (We discuss this case in greater detail later on.) Finally, an AGGREGATED direction 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 one-to-one and one-to-many constraints. However, subtleties regarding the propagation direction must be accounted for.

```
1 public void reportFailedPasswordAttempt(int userHandle) {
2    if(checkPermission(BIND_DEVICE_ADMIN) == GRANTED){
3       Binder.clearCallingIdentity();
4       policy.mFailedPasswordAttempts++;
5       if(policy.mFailedPasswordAttempts >= max))
6          if (userHandle == UserHandle.USER_OWNER) {
7             wipeDataLocked(wipeExtRequested, reason);
8          } else {
9             am.switchUser(UserHandle.USER_OWNER);
10            mUserManager.removeUser(UserHandle)
```

Listing 3: reportFailedPasswordAttempt

We explain further in the next example, which describes each step Poirot takes to generate observations and constraints from Listings 2 and 3.

(1) API resources `am.switchUser()` and `mUserManager.removeUser()` are reachable from API resource `reportFailedPasswordAttempt()` (O_1).
(2) InternalMethod resource `removeUserUnchecked()` is reachable from API `mUserManager.removeUser()` (O_1).
(3) API `mUserManager.removeUser()` is linked to permission `BIND_DEVICE_ADMIN` by a one-to-many constraint (R_2).
(4) InternalMethod resource `removeUserUnchecked()` is linked to permission `MANAGE_USERS` by a one-to-one constraint (R_1).

From (2) and (3), Poirot establishes a forward reachability constraint (R_3) to propagate the following:

(5) $AccessControl(\texttt{BIND_...ADMIN}, \texttt{mUserManager.removeUser()}, \texttt{FORWARD}) \xrightarrow{0.95} AccessControl(\texttt{BIND_...ADMIN}, \texttt{removeUserUnchecked()}, \texttt{FORWARD})$.

From (2) and (4), Poirot generates the following backward reachability constraint (R_5):

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

Similarly, from (1) and (6), Poirot derives the following backward reachability constraint (R_5):

(7) $AccessControl(p, \texttt{UM.removeUser()}, \texttt{BACKWARD}) \xrightarrow{0.95} AccessControl(p, \texttt{report...PasswordAttempt()}, \texttt{BACKWARD})$.

At this stage, the backward derived permission `MANAGE_USERS` for `reportFailedPasswordAttempt()` from `UM.removeUser()` can be *further propagated in a 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 a 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).

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 constraint (Rule R_{10}) to encode this observation. If a resource r_1 is control-dependent upon 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:

$$AccessControl(p, r_1, d^4) \xrightarrow{0.85} AccessControl(p, r_2^{write}, -)$$

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.

[4]We omit direction details for simplicity. More details can be found in Table 1.

Mutual Exclusivity. For this constraint, 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 discussed in Section 5.2). Note that the analysis avoids flagging error/validation checks, which 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 1:

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

5.2 Semantic Hints

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

Name Correlation. 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 3, we spot a naming similarity between API `reportFailedPasswordAttempt()` and field resource `policy.mFailedPasswordAttempt`. Hence, we can accordingly increase our confidence that the field access `policy.mFailedPasswordAttempt` requires `BIND_DEVICE_ADMIN`, which was initially assigned through a one-to-many control-dependency constraint.

To calculate the naming similarity between two resources a and r, Poirot relies on the DICE coefficient score [4]. It then establishes a naming correlation implication constraint between a and r if the DICE coefficient is substantially high. This constraint is depicted in Rule R_7, where direction $d \in \{\texttt{FORWARD}, \texttt{SELF}, -\}$:

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

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 upon p to exclude protections that may target 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 below in Rule R_9, 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 $<r^{getter}, r^{setter}>$ 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 constraints (Rules R_{12} and R_{13}), which propagate the getter's protections to the setter. Note that these constraints are unidirectional.

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

5.3 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 4.

```
1 String moveId;
2 public MigrationInfo getMoveData() {
3      if (checkPermissio("READ_MOUNT_DATA") == 0){
4         MigrationInfo info = new MigrationInfo();
5         info.moveId = moveId;
6         info.moveStatus = moveStatus;
7         return info;
8 public void moveData() {
9    moveId = readMoveData();
```

Listing 4: getMoveData

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

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 (Rule R_{14}):

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

Parameter Flow. We observe a special type of data-flow constraint that can help us refine the less certain one-to-many 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 constraints (Rules R_{15} and R_{16}):

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

5.4 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 1. Specifically, given a propagation direction d where d is not `AGGREGATED`, the confidence of $AccessControl(p, r, d)$ is faithfully propagated to $AccessControl(p, r, \texttt{AGGREGATED})$. 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.

6 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 [5] and relies on Akka Typed [1] to parallelize the analysis. We use ProbLog [3] as our 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 generates access control constraints. The analyzer implements a number of *Observation Extraction* modules, each responsible for identifying structural, semantic or data-flow observations. These observations are used to generate implication constraints in the form of Probabilistic Logic Program rules – i.e., $C \land x_1 \xrightarrow{p} x_2$. 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 7.2 for more detail.) We normalize the recommendations following [6] to allow comparison and effective inconsistency detection.

Table 3: Evaluation of APIs with High Confidence Access Control Recommendations

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

Example. We use the AOSP API `getSyncStatusAsUser()` defined in the `ContentService` to illustrate Poirot's output. The tool generates the three protection recommendations listed below. Note that the probabilities are enclosed in brackets.

(1) `INTERACT_ACROSS_USERS` ∧ `READ_SYNC_SETTINGS` **[0.91]**
(2) `INTERACT_ACROSS_USERS_FULL` ∧ `READ_SYNC_SETTINGS` **[0.91]**
(3) `INTERACT_ACROSS_USERS_FULL` ∧ `READ_SYNC_STATS` **[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, this case is considered consistent.

7 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.

7.1 (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 $CUTOFF$=0.90 and TOP_n=3. (Refer to Section 7.2 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 similarly named services at one time. The second additional setup considers three similarly named services at one time.

Results. Table 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 sharpen in-service probabilities.

7.2 (RQ2) Impact of Cut-off Criteria

This experiment evaluates the impact of the $CUTOFF$ and TOP_n criteria. Columns 3-6 in Table 4 report Poirot's accuracy using four TOP_n settings (namely, 1, 2, 3 and 4) and under three $CUTOFF$ configurations (0.85, 0.90 and 0.95). The last column reports the coverage achieved.

Table 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	0.90	76.6	76.7	77.4	77.4	59.4
	0.95	78.9	81.4	82.7	82.7	55.6

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 $CUTOFF = 0.90$ and $TOP_n = 3$ leads to an optimal trade-off between accuracy and coverage.

7.3 (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 0.80 to 0.90 and (2) the *Reachability* constraint with confidence varying from 0.50 to 0.60. As shown in Table 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%.

Table 5: Impact of Prior Probabilities

		Getter-to-Setter Constraint		
		p = 0.80	p = 0.85	p = 0.90
Reachability	p = 0.50	77.61	77.72	77.88
	p = 0.55	78.57	77.46	76.99
Constraint	p = 0.60	77.98	78.10	77.31

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.

7.4 (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, as the number of collected constraints plays a major role in the inference. To conduct this experiment, we examine AOSP using a similar setup to Experiment 7.3. We count and report the number of each constraint type found and present them in Figure 5.

In total, Poirot collects 2803 access control constraints and 3923 implication constraints from AOSP. Though all constraints contribute to the inference, the reachability and one-to-many constraints are particularly prevalent.

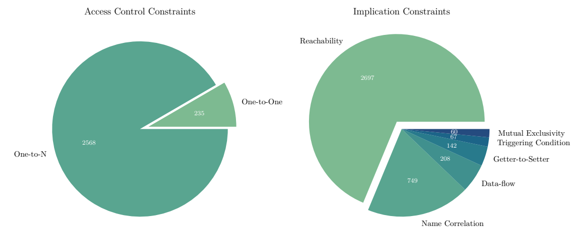

Figure 5: Breakdown of Probabilistic Constraints in AOSP

7.5 (RQ5) Runtime and Memory Overhead

Table 6 shows the execution time and memory consumption of Poirot on the analyzed ROMs. The results are broken down by analysis phase. Poirot's main bottleneck is the basic facts extraction, which relies on a path-sensitive, inter-procedural analysis. The execution time varies for different ROMs, taking more time for highly customized images, such as the Amazon Fire HD.

Table 6: Average Overhead Measurement*

ROM	Basic Fact Extraction		Implication Constraint Generation		Probabilistic Inference		Inconsistency Analysis	
	Time	Memory	Time	Memory	Time	Memory	Time	Memory
AOSP	50.0	332.3	22.8	302.1	23.5	367.5	2.7	365.4
Xiaomi Poco C3	53.3	373.1	33.4	280.8	30.3	361.5	4.1	366.3
Amazon Fire HD	56.0	301.9	32.4	309.3	31.0	320.3	4.5	382.3
LG LM-V405	54.1	327.1	31.4	300.9	28.0	338.2	3.3	367.4

*Time is in minutes, memory in mB.

7.6 (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 7.

Experiment Setup. Unlike Experiment 7.1, 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.

Results. Table 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 [5] 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.

Inconsistencies Uniquely Discovered by Poirot. Column 5 in Table 7 lists the number of inconsistencies that were detected using at least one non-reachability implication constraint.

[5]This case was discussed in the Motivation section and illustrated in Figure 2.

Table 7: Poirot's Inconsistency Detection Results

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

As shown, 10 inconsistencies were uniquely detected 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.

7.7 (RQ8) Suppressing False Positives of Other Tools

This experiment assesses whether Poirot successfully suppresses the high false positives seen in Kratos [22] and AceDroid [6], two state-of-the-art access control inconsistency detection tools. Both tools 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.

Results. Table 8 reports the results. As shown, both AceDroid and Kratos+ generate substantial FPs ranging from 71% to 81% in AceDroid and from 84% 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.

Table 8: AceDroid and Kratos+'s False Positives

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	62 (84.9)	61.1

False Positives Suppression by Poirot. As shown in Columns 4 and 7 in Table 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.

8 CASE STUDIES

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

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

Table 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
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
LG LM-V405	LGMDM. getRunningPackagesFromPid	UserCheck	UserCheck AND (REAL_GET_TASKS)	Data Flow Reachability	Exfiltrate running packages details
Fire HD 10	AmazonInput.setInputFilter	E	SYSTEM_PERMISSION	Reachability Naming Correlation	Key Logger
Fire HD 10	MigrationService.migrate	Normal_Permission	MOVE_PACKAGE	Setter-getter Naming Correlation Forward Reachability	Local system crash Reboot
Fire HD 10	MigrationService.getMigrateData	Normal_Permission	MOVE_PACKAGE	Data Flow	Obtain migration meta data
Fire HD 10	AmazonPMS.setAmazonFlags	E	SYSTEM_PERMISSION	Trigger Condition	Change Amazon-Specific package settings*
Fire HD 10	AmazonPMS.removeAmazonFlags	E	SYSTEM_PERMISSION	Trigger Condition	Change Amazon-Specific package settings
Xiaomi Poco C3	IPerfShielder. getAllRunningProcessMemInfos	E	UserCheck AND (REAL_GET_TASKS)	Reachability	Exfiltrate running processes info**

*Amazon mentioned that they have fixed the vulnerability thanks to an earlier report.
**The vendors have acknowledged the issues but mentioned that the cases were known internally/reported before us.

`migrate()` was subsequently flagged as an inconsistency due to a weaker protection enforcement.

We tested the reported vulnerability and confirmed that both APIs lack protections. Triggering `migrate()` with specific parameters (i.e., supplying private data folder to be migrated) crashes the system server.

9 RELATED WORK

Probabilistic Program Analysis. Probabilistic type inference [28] has been proposed for dynamic programming languages, e.g., Python. Probabilistic model checking [13, 16, 20] enhances the existing deterministic techniques by encoding probabilities into the transition among states. With largely extended scalability, probabilistic symbolic execution [10, 17] efficiently predicts the likelihood of reaching a certain program point. Researchers also adapt probabilistic inference and distribution analysis techniques in the domain of binary analysis [21, 31] to provide a systematic approach to model the inherent uncertainty caused by information loss during compilation. Other applications include fuzzing [32], network trace analysis [30], race and leak detection [11, 19] and runtime event analysis for program understanding [25, 33]. To the best of our knowledge, Poirot is the first approach to leverage probabilistic analysis to generate Android access control recommendations.

Security Property Inference. Inference techniques have been widely adopted for vulnerability detection and security invariant validation. Engler et. al. [14] devise a static checker to infer bugs in real systems such as Linux and OpenBSD. AutoISES [24] automatically infers high-level security specification and detects violation afterwards. Srivastava et. al. [23] adapt a precise, flow and context-sensitive security policy inference technique to analyze relationships between security checks and security-sensitive events. Vaughan et. al. [26] devise a security-expressive language to describe security policy where inference of expressive is introduced to help reduce the number of annotations. JIGSAW [27] infers programmer expectations to achieve better access control. Yamaguchi et. al. [29] leverage inference techniques to search taint-style vulnerabilities in C code. Inspired by these works, Poirot adopts rule inference techniques to recommend Android access control using probabilistic constraints, which naturally model the uncertainty inherent in statically extracted API-to-protection mappings.

Inconsistency Detection. Inconsistency detection tools pinpoint security policy inconsistencies. Two recent works extend their scope beyond the Android framework. FReD [2] identifies inconsistencies in API access control requirements by analyzing Linux-layer permissions. IAceFinder [34] detects cross-context inconsistencies in the Java and native layers. Kratos [22], AceDroid [6] and ACMiner [18], discussed elaborately throughout the paper, leverage in-framework security oracles.

API-to-Protection Mappings. Stowaway [15] and Dynamo [12] deduce permission requirements for Android APIs using a dynamic approach. PScout[8], Axplorer[9] and Arcade [7] address the same issue using static analysis. Similarly, Poirot statically infers protection recommendations; however, it accounts for inherent uncertainty using a probabilistic approach.

10 CONCLUSION

We propose Poirot, a novel probabilistic access control recommendation framework for Android resources. The framework features tailored static analysis to collect various implicit relations beyond reachability that can connect resources and access control. The relations are organically transformed into implication constraints to connect resources with inherent uncertainty and predict their protection recommendations. We applied our framework to analyze four Android ROMs. Our evaluation shows that Poirot effectively generates protection recommendations and detects inconsistencies.

ACKNOWLEDGMENTS

We wish to thank Güliz Seray Tuncay for kindly providing her feedback on our collected rules. This research was supported in part by NSERC under grant RGPIN-07017, by the Canada Foundation for Innovation under project 40236 and by a Google ASPIRE award. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo. Any opinions, findings and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

- [1] 2022. Akka. https://akka.io/
- [2] 2022. FReD: Identifying File Re-Delegation in Android System Services. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/gorski
- [3] 2022. ProbLog. https://dtai.cs.kuleuven.be/problog/
- [4] 2022. Sorenson-Dice Coefficient. https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient
- [5] 2022. WALA. https://github.com/wala/WALA
- [6] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. Internet Society. https://doi.org/10.14722/ndss.2018.23121
- [7] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise android API protection mapping derivation and reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 1151–1164. https://doi.org/10.1145/3243734.3243842
- [8] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 12. PScout: Analyzing the Android Permission Specification. In *CCS*. 1070.
- [9] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 48.
- [10] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Pasareanu. 2015. 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*. 866–877.
- [11] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. 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*. 810–821.
- [12] Abdallah Dawoud and Sven Bugiel. 2021. Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework, In Network and Distributed Systems Security (NDSS) Symposium 2021. *Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework*. https://publications.cispa.saarland/3340/
- [13] Alastair F. Donaldson, Alice Miller, and David Parker. 2009. 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*. 289–298.
- [14] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72.
- [15] Adrienne Porter Felt. 2011. Permission Re-Delegation: Attacks and Defenses. In *20th USENIX Security Symposium (USENIX Security 11)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/usenixsecurity11/permission-re-delegation-attacks-and-defenses
- [16] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2011. 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*. 341–350.
- [17] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 166–176.
- [18] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. (1 2019). http://arxiv.org/abs/1901.03603
- [19] Matthias Hauswirth and Trishul M. Chilimbi. 2004. 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*. 156–164.
- [20] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. 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*. 585–591.
- [21] Kenneth A. Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1187–1198. https://doi.org/10.1109/ICSE.2019.00121
- [22] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2017. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. Internet Society. https://doi.org/10.14722/ndss.2016.23046
- [23] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. 2011. A security policy oracle: Detecting security holes using multiple API implementations. *ACM SIGPLAN Notices* 46, 6 (2011), 343–354.
- [24] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations.. In *USENIX Security Symposium*. 379–394.
- [25] Neil Toronto, Jay McCarthy, and David Van Horn. 2015. 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*. 53–79.
- [26] Jeffrey A Vaughan and Stephen Chong. 2011. Inference of expressive declassification policies. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 180–195.
- [27] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. 2014. {JIGSAW}: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium (USENIX Security 14)*. 973–988.
- [28] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. 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*. 607–618.
- [29] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.
- [30] Yapeng Ye, Zhuo Zhang, Fei Wang, Xiangyu Zhang, and Dongyan Xu. 2021. 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. https://www.ndss-symposium.org/ndss-paper/netplier-probabilistic-network-protocol-reverse-engineering-from-message-traces/
- [31] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. 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*. IEEE, 813–832. https://doi.org/10.1109/SP40001.2021.00051
- [32] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*.
- [33] Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*. 91–100.
- [34] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. 2022. Uncovering Cross-Context Inconsistent Access Control Enforcement in Android.