USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Ariadne: Navigating through the Labyrinth of Data-Driven Customization Inconsistencies in Android

Parjanya Vyas, Haseeb Ur Rehman Faheem, Yousra Aafer,
and N. Asokan, *University of Waterloo*

https://www.usenix.org/conference/usenixsecurity25/presentation/vyas

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

Ariadne: Navigating through the Labyrinth of Data-Driven Customization Inconsistencies in Android

Parjanya Vyas
University of Waterloo
parjanya.vyas@uwaterloo.ca

Haseeb Ur Rehman Faheem
University of Waterloo
hfaheem@uwaterloo.ca

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

N. Asokan
University of Waterloo
asokan@acm.org

Abstract

Vendor customization of the Android framework is known to introduce security concerns. One type of customization is *data-driven*, involving changes to access-controlled framework variables, which we call *data holders*. Analyzing the security of data-driven customization has not been explored in prior work because it faces several challenges as it requires modeling implicit access control (AC) relations among Java objects and their corresponding operation semantics. Existing Android AC inconsistency detection approaches struggle to discover data-driven AC inconsistencies.

We propose a novel approach, Ariadne, to address these challenges by (1) constructing an abstract representation, the *AC dependency graph*, to model AC relationships among framework data holders, and (2) using it to detect missing AC enforcement in data holders and their corresponding APIs. Using two AOSP and 11 custom ROMs, we show that Ariadne detects 30 unique data-driven AC inconsistencies which *cannot be detected by existing approaches*. Therefore Ariadne can offer more comprehensive protection by effectively complementing existing AC inconsistency detection approaches.

1 Introduction

The openness of Android allows device vendors to customize the codebase, making various, often extensive, changes. The customization at the framework-level, the software stack that implements system services, can involve substantial changes to framework APIs to implement custom functionality.

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

Data-driven customization risks introducing vulnerabilities like AC anomalies (e.g., an under-protected new data holder, or a data holder with inconsistent AC across different APIs), which malicious app developers could exploit. The security implications of data-driven customization, however, are largely unexplored. Data-driven inconsistencies are challenging to detect/mitigate using existing tools, as they exhibit unique patterns not tackled by prior work [2, 16, 18, 32, 39, 50].

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

In our study of 11 custom ROMs, we found that data-driven customization is highly prevalent: e.g., 45027 instances found in Samsung ZFlip v13 (§7). Their complexity and *implicit AC implications* make them potential attacker targets.

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

often non-existent, security specifications.

Data holders have two kinds of implicit "*security specifications*"[1] that may be overlooked during customization: those implied by (i) Java objects semantics (e.g., an AC-protected field implies that the containing class object should also be protected - e.g., `TelephonyManager` has a data holder `mIccRecords` of type `IccRecords`. An AC protected field `mImei` of class `IccRecords` implies that the `mIccRecords` object should also be protected), or (ii) supported operation semantics (e.g., AC for an operation that checks if an entry exists in a collection implies that reading the entry should also be protected - if `hasIccCard` from `TelephonyManager` is protected, then `getIccCards` should also be protected). These implications can be non-deterministic (e.g., an AC-protected class object may or may not imply that one of its member fields should be protected). To effectively utilize non-deterministic implications, we must model their associated *uncertainty* accurately.

Prior inconsistency detection approaches in Android either 1) rely on convergence [2, 16, 33] or 2) utilize uncertain *hints* such as control-dependency, naming similarity, or UI based security indicators [11, 39]. However, none of them can reason about implicit security implications in data holders and hence cannot accurately catch underlying data-driven inconsistencies. The diversity in data holders necessitates modeling of their structure, internal fields and relations among them, involving unique challenges (§2) that require dedicated analysis: (1) converting the diverse data types of framework data holders into a cohesive, unified abstraction for effective analysis, and (2) using ambiguous AC implications connecting data holders. Existing tools do not support such modeling. We present a new tool, Ariadne, which addresses these challenges as follows:

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

Access Control Inference and Inconsistency Detection. Ariadne leverages the statically extracted AC information and deterministic edges to *annotate* nodes with required AC. The generated graph is often partially annotated. To predict missing annotations, Ariadne uses the non-deterministic edges. Specifically, it treats the graph as a flow network [30] and uses flooding [4] to transitively propagate and predict AC annotations along the labeled edges (§6.3). Finally, data-driven inconsistencies are detected when the predicted AC annotations are stronger than the implemented ones.

Results. With Ariadne, we performed the *first systematic study* of data-driven customization in Android using two AOSP and 11 vendor-customized ROMs. On AOSP, Ariadne predicts correct AC in 553 (97%) of data-driven related APIs. On custom ROMs, from seven vendors, it detects 90 unique data-driven inconsistencies, with a manually estimated false-positive (FP) rate of 11.8%. To demonstrate the impact of the inconsistencies, we built end-to-end exploits for 13 cases with various consequences; including enabling/disabling arbitrary critical packages in Tecno, launching random Activities with system privilege in Vivo (an unauthorized app can trigger system-protected Activities such as *factory resetting the device*) and altering critical data policies in Samsung.

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

We summarize our contributions as follows: We

- identify the hitherto understudied problem of access-control inconsistencies in the Android framework resulting from *data-driven customization*, (§2)
- propose Ariadne. (§4,§5,§6), a novel approach we use for the *first systematic study* of Android data-driven customizations, showing Ariadne's effectiveness on 11 custom Android ROMs, discovering 90 unique access control inconsistencies, (§7) and
- show how Ariadne complements existing techniques by detecting 30 unique new inconsistencies, not caught by other tools (ACMiner, Poirot and Bluebird). (§8)

2 Motivation

Prior work has proposed detecting AC flaws in the Android framework using a range of techniques such as convergence-based inconsistency detection as in Kratos [32], IAceFinder [50] and probabilistic inference as in Poirot [11] and Bluebird [39]. However, these are limited, if not incapable, in their ability to catch *data-driven inconsistencies*.

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

2.1 Java Objects Access Control Semantics.

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

[1] Following [39], we use "security specification" in a broad sense, to refer to the security policy inferable from language and data types semantics.

Example. To illustrate this insight, we use `SemTelephonyController`, a custom system service defined by Samsung to tailor AOSP's Telephony framework. Figure 1 shows its UML class diagram, simplified by omitting some data types and methods for clarity. `SemTelephonyController` uses a number of AOSP data types (shown in green) and customizes some (shown in red). The customization entails adding new (i) fields like `mEuimid` and (ii) utility methods like `getEuimid`.

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

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

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

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

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

Objective-sensitivity is required to associate AC with the right *object* of a class. As various APIs may access different instances of the same class, object-insensitive analysis can lead to overly restrictive AC inference. This may result in false positives – we illustrate this through a case study in §9.

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

TC3: Diverse data types. Data holders may follow diverse hierarchical structures. For example, classes can define nested classes, child classes can have their own set of fields while still inheriting other fields from the parent class. To detect AC implications, the analysis should be able to (1) represent the complex hierarchical structure and (2) trackAC relations among elements.

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

2.2 Operation Access Control Semantics

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

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

Listing 1: getExtensions and hasExtension APIs

Example. Xiaomi defines two APIs `getExtensions` and `hasExtension` in a custom system service, `SecureSpacesService` (Listing 1). The two APIs operate on `mExtensions`, a HashMap that maps a String key to a value of a custom composite data type *SecureSpaceExtension* (Line 11-Line 13). The APIs reflect different operations performed on the data holder `mExtensions`. `getExtensions` is a generic getter as it returns *all* keys from `mExtensions` (Line 3) whereas `hasExtension` checks for the existence of a key in `mExtensions` (Line 8). `hasExtension` enforces a system-level AC (Line 7)

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

TC5: Modeling operation semantics. The analysis should account for operation semantics; otherwise, inconsistencies like the one in Listing 1 would go undetected. Besides, the analysis should infer the operation granularity, if any – which may require modeling developer-implemented logic in APIs.

3 Background

To provide context for our approach, we first summarize key AC issues in Android and relevant inconsistency analysis techniques. Two central AC issues analyzed by prior work are (1) inconsistencies stemming from the decentralized placement

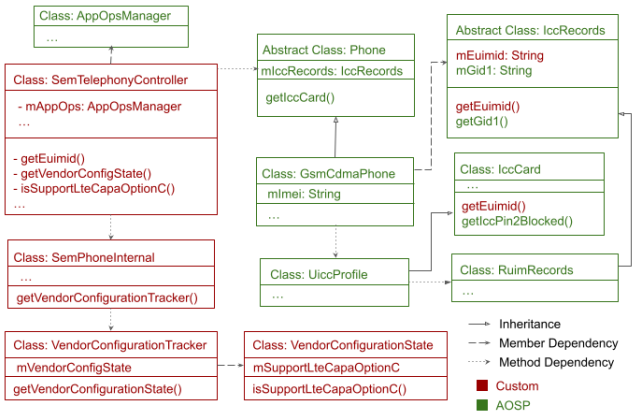

Figure 1: `SemTelephonyController` UML Class Diagram

of AC checks across APIs and framework layers (e.g., Java and native code), and (2) the absence of formal AC specifications for framework APIs.

To address the AC inconsistencies (issue 1), recent research has focused on using differential analysis techniques. One such technique is *convergence analysis*, which aims to compare AC enforcement within different code paths leading to the same sensitive resource. The AC being compared can either be within the same layer [2, 32] or across different layers [18, 50]. If the AC enforcement within different paths leading to the same resource differs, then such an inconsistency may lead to an exploitable vulnerability.

The absence of formal security specifications for the APIs (issue 2) leads to AC decisions concluded from developers' subjective assessments of an API's sensitivity. As a result, the correct security implementation is often ambiguous and uncertain. This uncertainty is the main cause behind the high false positive rate of pure convergence-based techniques. Recent works [11, 39] address this challenge through probabilistic inference. The works rely on collecting and aggregating various hints pertaining to resource sensitivity to conclude a probabilistic AC specification. Probabilistic inference approaches proved to be more effective and precise than convergence-based techniques, by introducing less false positives and uncovering more vulnerabilities.

4 Approach Overview

To address the challenges in §2, we propose Ariadne, a systematic approach for inferring AC inconsistencies caused by data-driven customization. Figure 2 shows Ariadne's two-stage workflow.

Stage 1: Modeling via AC Dependency Graphs. Given an Android ROM, Ariadne performs static analysis (Step 1) on defined system services[2] to identify reachable framework variables and corresponding type information. To address TC1 and TC2, Step 1 performs *selective field- and object-sensitive* analysis to recover a set of *target variables* that are potential data holders, chosen based on the criteria discussed in §5.1.

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

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

Step 3 annotates nodes in the graph. The collected AC annotations may only cover those nodes reachable from protected APIs. Hence, Step 3 may generate a *partially-annotated* graph. We predict missing AC annotations as follows: (1) Automatically propagate AC annotations between nodes connected via *deterministic* edges. (2) Use a custom flooding [4] algorithm that can - (i) allow quantifying the uncertainty, and (ii) conditionally propagate AC annotations to the nodes connected via *non-deterministic edges*, as described next.

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

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

To determine the set of possible annotations for sinks, Ariadne begins by assigning each AC annotation in the sources an initial weight, *relevance_weight*, denoting our initial belief of the AC annotation being *relevant* to the nodes. It then uses a flooding [37] algorithm to adjust the *relevance_weight* based

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

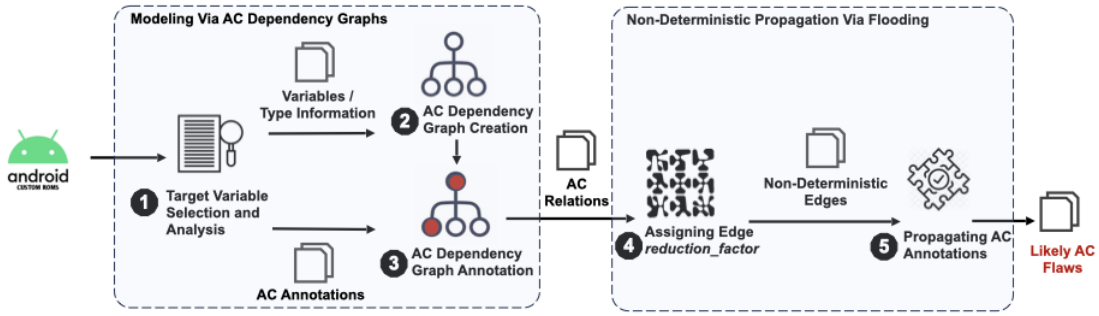

Figure 2: Ariadne System Diagram

on connected nodes. The newly updated *relevance_weight* reflects an aggregated value computed based on the (number of, and types of) edges connecting the node to other nodes.

Ariadne assigns a *reduction_factor* to each edge based on its initial labels, e.g., an edge connecting a parent class to a member field has a *reduction_factor* of 0.6 (meaning that the member field has a 60% chance to inherit its parent's AC annotations). The flooding algorithm then uses the *reduction_factor* for controlled propagation of new annotations, adjusting their *relevance_weight* along the way. When multiple incoming AC annotations flow into a sink, Ariadne combines their *relevance_weight* using the noisy OR [35] operation. We describe propagation further in §6.3.

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

5 Modeling via AC Dependency Graphs

5.1 Step 1: Target Variable Selection

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

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

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

We only focus on the first. Analyzing non self-contained types, though more comprehensive, has two major disadvantages: (1) AC inconsistencies in non self-contained types are unlikely to be caused by data customization (§7.2), and (2) it is more expensive as it generates highly-complex AC dependency graphs. By leaving them out, we risk false negatives, which are very unlikely as we show later (§7.2).

Ariadne collects **target variables** by extracting system services similar to prior work [2, 6, 16] and conducting reachability analysis to extract reachable classes; i.e., those instantiated or statically accessed by the service. Each identified class is then inspected to extract corresponding instance and static fields declared by the class or by any of its parent classes. Fields not conforming to the selection criteria above are excluded from the analysis.

5.2 Step 2: Graph Creation

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

Graph Node denotes a target variable. It stores the variable name, data type, defining class, and instantiating class. We divide graph nodes into three categories: (1) *Simple* nodes encompass Java primitive types and certain Android-specific types such as `Bundle` and `Parcel`. They correspond to the leaf nodes in the graph because they cannot have member fields. (2) *Complex* nodes represent types with one or more member fields, hence they have one or more outgoing edge(s). (3) *Collection* nodes represent a unified type for storing and manipulating a group of Simple, Complex, or Collection types. In addition to Java types extending the `Collection` interface, we also consider `Maps` and `Arrays` as Collection types. Typi-

Table 1: Relations Between AC Dependency Graph Nodes

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

cally, `Collection` nodes should have an outgoing edge for each element in the collection; however, due to their dynamic nature, it is often difficult to resolve the number statically. Instead we create two special outgoing edges that connect a `Collection` node to two synthetic nodes, `ALL` and `SOME`, representing two possible member (access) granularities. `ALL` denotes the granularity where *all* members of the collection are accessed; whereas `SOME` represents accesses to a subset.

Directed Edges encode relations connecting pairs of nodes. Each edge label corresponds to one of the relations in Table 1.We group the relations into two classes, *deterministic* and *non-deterministic*. Deterministic relations reflect an *always true* AC implication – if two nodes are connected via a deterministic edge, their AC annotations should be equivalent. For example, an edge from a member field node to its containing class object node is deterministic, hence the field's AC annotation can be propagated to the containing object node.

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

Graph Construction (Algorithm 1) is our process for constructing AC dependency graph(s) from a given Android system service. It takes the set of target variables *targetVars* generated in Step 1 (§5.1) and their resolved concrete types, and produces the corresponding AC dependency graphs.

To resolve the concrete types of variables, we track each new object-allocation site (e.g., through identifying allocation instructions such as `SSANewInstruction`). Different instances of the same type will each have their own nodes. Objects extending the same super-class will be similarly allocated dedicated nodes. This way, the AC dependency graph ensures that tracking occurs at object-instance level (nodes represent unique object instances). We presented a case study in §2.1 (Listing 3) to illustrate the importance of object-sensitivity.

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

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

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

In line 35, the algorithm connects two AC dependency graphs. This happens when a node is found to be connected to a node belonging to an already constructed graph (e.g., when two APIs in a system service operate on two different

Algorithm 1: Construct an AC dependency graph

```
   Input    : targetVars
   Output   : ACDepGraphNodes
   Requires :
     1. getNodeCat(var) : Returns the node category for var.
     2. createNode(nodeCat, ID, type, struct) : Creates a new node of category
        nodeCat .
     3. addEdge(n1, n2, edgeType) : Connects ordered nodes n1 and n2
                with an edge labeled edgeType.
     4. getInnerCollectionType(type) : Returns the Java type of members of a
        Collection.
     5. getNodes(type, struct) : Returns a set of all nodes with Java type type and
        declaring class struct.
     6. getMembers(type) : Returns a set of all member fields of class type.
 1 ACDepGraphNodes=[]
 2 for tVar ∈ targetVars do
 3     ID = tVar.ID, type = tVar.type, struct = tVar.getDeclaringClass()
 4     nodeCat = getNodeCat(tVar)
 5     node = ϕ
 6     switch nodeCat do
 7         case SIMPLE do
 8             node = createNode(SIMPLE, ID, type, struct)
 9         end
10         case COMPLEX do
11             node = createNode(COMPLEX, ID, type, struct)
12             memberNodes = []
13             for m ∈ getMembers(type) do
14                 memberNode = createModel(m)
15                 addEdge(node, memberNode, containsMember)
16                 memberNodes.add(memberNode)
17             end
18             for (c1, c2) ∈ childNodes do
19                 addEdge(c1, c2, siblingOf)
20             end
21         end
22         case COLLECTION do
23             node = createNode(COLLECTION, ID, type, parent)
24             innerType = getInnerCollectionType(type)
25             someNode = createModel(innerType)
26             someNode.ID = SOME
27             allNode = createModel(innerType)
28             allNode.ID = ALL
29             addEdge(node, someNode, collectionOf)
30             addEdge(node, allNode, collectionOf)
31             addEdge(someNode, allNode, generic_specific)
32         end
33     end
34     for insNode ∈ getNodes(type, struct) do
35         addEdge(node, insNode, shareSuperType)
36     end
37     ACDepGraphNodes.add(node)
38 end
39 return ACDepGraphNodes
```

members of the same target variable. As shown, the algorithm constructs the graph recursively repeating these steps for each member type (line 15, line 25, line 27).

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

Listing 2: Inconsistency in `SecureSpacesService`

Figure 3 illustrates the AC dependency graph corresponding to the target variable `mExtensions` in Listing 2. Next, we walk through this example to describe how Algorithm 1 creates the AC dependency graph:

- The target variable `mExtensions` is determined to be a *COLLECTION* (line 4). Therefore, two synthetic child nodes, `ALL` and `SOME`, are created and connected to the parent `mExtensions` using a *collectionOf* edge (line 29 - line 30). (nodes omitted from the figure for brevity[3])
- The inner type of `mExtensions` is determined to be `<String, SecureSpacesExtension>` (line 24). The String key is determined to be a *SIMPLE* category. A corresponding node will be created (not shown for brevity).
- `SecureSpacesExtension` is determined to be a *COMPLEX* category. The algorithm resolves its member fields and creates corresponding nodes. One such node represents the member `userRestrictions` which is in turn determined to be of a *COLLECTION* (line 4).
- The process similarly creates two synthetic members, `ALL` and `SOME` nodes of inner type `UserRestriction`, and connects them to the parent `userRestrictions` using *collectionOf* edge (line 29 - line 30).
- `UserRestriction` is determined to be of *COMPLEX* category (line 4). Two nodes, representing its member fields *name* and *deviceOwnerOnly*, are then created and connected to the containing class object node `UserRestriction` using *containsMember* edge (line 15). The construction finishes at this stage since there are no more nodes to be resolved.

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

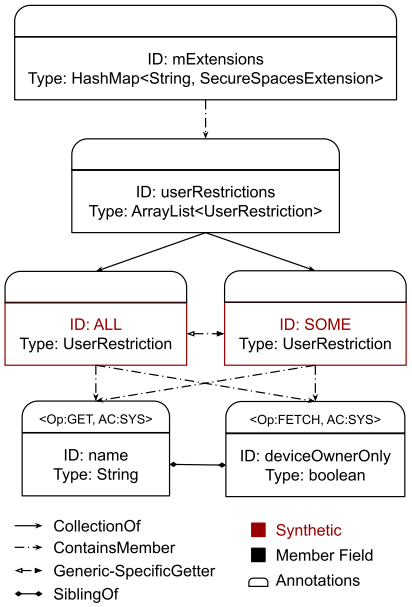

Figure 3: AC Dependency Graph for `mExtensions`

5.3 Step 3: Annotation

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

Target Operations and Granularity. Utility APIs operating on composite data types provide various operational semantics (e.g., check if an element is in a Collection, return an element in a Collection) which may carry AC implications (§2). Modeling the entirety of operation semantics is infeasible to develop (given the domain-specific nature of Android APIs) and even to analyze (unclear AC implications). Hence, we focus our analysis scope on a predefined set of operation semantics that can have*intuitive and direct* AC implications[4].

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

AC Extraction. Ariadne collects AC-related information using path-insensitive analysis. It performs a forward control-flow analysis on the API's inter-procedural control flow graph (ICFG) and pinpoints the conditional branches on which *a target sink* is control dependent – where a target sink corresponds

[4]These include both certain and uncertain implications.

Table 2: Operation and Granularity Semantics

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

to an operation performed on a target variable. The analysis follows the traditional approach proposed by Kratos [32] and Axplorer [7] to resolve and identify conditional and non-conditional statements traditionally used for AC enforcement. Finally, the identified AC checks are normalized using the approach proposed by AceDroid [2]; where each AC check is categorized based on the level of permission it requires: (i) Normal, (ii) Dangerous, (iii) Signature, and (iv) System. AC checks that do not rely on permissions are categorized as System since they check for a privileged user.

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

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

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

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

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

6 Non-deterministic Propagation via Flooding

To predict missing AC annotations, we treat the AC dependency graph as a *flow network* [26, 30], using controlled flooding to propagate annotations through non-deterministic edges.

6.1 Preliminaries

Flow networks in data systems [26, 30] are directed graphs representing systems where resources (data) transfer through edges. *Source* nodes initiate flows with an initially assigned *flow rate*. *Sink* nodes consume incoming flows, possibly from multiple sources. If the capacity of an edge is smaller than the flow rate of a data flow, then data flow is prevented. An extension of traditional graphical flow networks with multi-commodity flows [25] allows multiple independent flows to traverse the same network.

Ariadne treats the AC dependency graph as a flow network where AC annotations flow across connected nodes. Nodes with deterministic annotations (in Stage 1) act as sources producing annotations. Since these are inferred through program analysis, Ariadne assigns them an initial high flow rate (*relevance_weight*) denoting that the annotation is highly relevant. Ariadne then (transitively) propagates the annotations via non-deterministic edges while adjusting the *relevance_weight* by applying an edge's capacity (*reduction_factor*) which expresses the uncertainty of its AC implication.

Ariadne calculates the *relevance_weight* of the source's AC annotation to the sink as follows: $relevance\_weight(sink, source\_ac\_annotation) = relevance\_weight(source, source\_ac\_annotation) \times reduction\_factor$ where $reduction\_factor \in [0.4, 0.95]$ is associated with each non-deterministic edge type (Table 5).

Flooding is a technique for propagating flows through a flow network. In the basic flooding approach, each node forwards received flows to its neighbors, thereby guaranteeing that every connected node eventually receives the flow. To mitigate issues arising from redundancy, variations such as controlled flooding [4] and probabilistic flooding [15] selectively forward flows to a chosen set of neighboring nodes.

We use a controlled flooding [28] method to propagate AC annotation flows through non-deterministic.

6.2 Step 4: Assigning edge *reduction_factor*

Table 3: Formal Notations and their Definitions

Term	Notation and Definition
Protection	$p \in \{\text{NONE}, \text{NORMAL}, \text{DANGEROUS}, \text{SYS\_OR\_SIG}\}$[5]
Operation	$op \in \{\text{GET}, \text{SET}, \text{ACCESS}, \text{MODIFY}, \text{ADD}, \text{REMOVE}, \text{IND\_EXISTS}, \text{VAL\_EXISTS}\}$
AC dependency graph nodes	*GN*
AC dependency graph edges	*GE*
AC(n, p, op)	Node $n \in GN$ requires `protection` p for `operation` op
relevance_weight (n, p, op)	*relevance_weight* of AC for node n

Table 3 presents the notations we use to define the non-deterministic edge labels (Table 5). `Protection` p denotes a normalized form of Android AC [6]. We use op to denote an `Operation`, which refers to the operation semantics (Table 2).

[5] We borrow the ordering of AC from [2] as: NONE < NORMAL < DANGEROUS < SYS_OR_SIG

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

Table 4: Definitions of AC Relations

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

Table 5 shows the *reduction_factor* for each type of *non-deterministic* edge. Each row lists the condition(s) that identifies a *non-deterministic* edge.Table 4 lists the AC relations that correspond to *the non-deterministic* edges.

Logical relations. Edges propagate AC annotations by relying on Java object AC relations. Edges 1 and 2 state that AC annotations of a class instance node n_1 can be propagated to its member field node n_2 with *reduction_factor* rf. If n_1 is the only member of n_2 (1), we link the AC implications with a high confidence, hence $rf = 0.95$. Otherwise, if n_2 has more than one member fields (2), we are less certain about the AC implication, hence $rf = 0.6$. Edge 3 identifies an edge between two sibling nodes, and sets $rf = 0.6$ given the uncertainty of the AC relation. Edge 4 states that nodes whose types extend the same superclass may share a similar protection. The edge is bidirectional with $rf = 0.7$.

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

For example, two sibling nodes with similar variable names are more likely to have similar AC annotations. Similarly, a member field node is likely to inherit its containing class node's protection if their names are similar. We introduce the edge in Edge 5 to encode this observation. The *reduction_factor* is a function of $sim(n_1, n_2)$, a similarity score of n_1 and n_2. We calculate the similarity score using four features: (i) variable names, (ii) types, (iii) associated modifiers (e.g., public, private) and keywords (e.g., static, volatile), and (iv) names of methods operating on the field. (§A.1)

Granularity semantics relations. These edges allow inferring AC based on the operation granularity. Particularly, the relations *Specific-to-generic* (for *Collection* types) imply that if a subset of the collection members requires an AC, then the collection itself requires a similar AC. The opposite implication is uncertain. In our graph representation, this relation corresponds to the edge connecting <ALL, SOME> (§5.2).

Ariadne identifies edges between nodes n_1 and n_2 of type ALL and SOME as depicted in Edge 6 and 7, where the AC implication is propagated with $rf = 0.95$ from n_1 to n_2 and with $rf = 0.4$ in the opposite direction.

6.3 Step 5: Propagating AC annotations

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

Cross-nodes AC propagation assigns AC annotations of nodes (line 4). It adjusts corresponding *relevance_weight* based on the *reduction_factor* of the edges connecting them. line 4 combines *relevance_weight* of multiple AC annotation flows from incoming edges using a noisy OR operation [35], ensuring that the combined value is less than 1.

In-node AC propagation propagates AC annotations across different operations within the same node (Table 2). This allows to express AC implications between common operations. For example, an AC annotation for a *GET* operation implies that the *MODIFY* operation likely requires an annotation that is at least as strong.

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

Output of the algorithm is a list of AC annotations per operation, ranked by *relevance_weight*, for each node in the AC dependency graph. Ariadne then propagates annotations to corresponding APIs (that is, those operating on data holders denoted by the nodes). We use an empirically-chosen *cut-off* value (§7.6) to select highly relevant AC annotations (i.e., where *relevance_weight* > *cut-off*).

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

Algorithm 2: AC Propagation through Flooding

Input : *ACDepGraph* with node set = *GN*
Output : Set of final *relevance_weight* values for annotations of data holders
Requires :
1. neighbours(n): Returns the list of nodes connected to n in *ACDepGraph*.
2. flood(n_1, n_2): $\forall AC \in$ annotations of n_1, $relevance\_weight(n_2, p, op) = relevance\_weight(n_2, p, op) \oplus relevance\_weight(n_1, p, op) \times reduction\_factor$.
3. allOps(n): A list of all operations performed on node n.
4. $op\_sem(op_1, op_2)$: Returns *reduction_factor* corresponding to AC implication between operations op_1 and op_2.

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

Table 5: Edge definitions and their *reduction_factor* values

Edge ID	Edge Label	Condition	*reduction_factor* (rf)
1	Class-1-Member	$n_1 \in GN \wedge n_2 \in GN \wedge contains(n_1,n_2) \wedge one\_member(n_1)$	0.95
2	Class-m-Members	$n_1 \in GN \wedge n_2 \in GN \wedge contains(n_1,n_2) \wedge m\_members(n_1)$	0.6
3	Siblings	$n_1 \in GN \wedge n_2 \in GN \wedge sibling(n_1,n_2)$	0.6
4	shareSuperType	$n_1 \in GN \wedge n_2 \in GN \wedge same\_supertype(n_1,n_2) \wedge n_1 \neq n_2$	0.7
5	Similarity	$n_1 \in GN \wedge n_2 \in GN \wedge (contains(n_1,n_2) \vee sibling(n_1,n_2))$	$0.6 \times sim(n_1,n_2)$
6	Specific-Generic	$n_1 \in GN \wedge n_2 \in GN \wedge specific\_generic(n_1,n_2)$	0.95
7	Generic-Specific	$n_1 \in GN \wedge n_2 \in GN \wedge specific\_generic(n_2,n_1)$	0.4

7 Implementation and Evaluation

We develop a prototype of Ariadne consisting a static analyzer, built using WALA [40], and Java based implementations of Algorithm 1 and Algorithm 2. Our analysis is performed on a machine with an 8-core CPU (Intel(R) Core-i7-10510U @ 1.80GHz) and 16G main memory.

7.1 Test ROMs

To test our prototype, we analyze 13 ROMs, collected from public online repositories [1, 14] and physical devices. The samples range from Android versions 8 to 14 and correspond to two AOSP ROMs and 11 custom ROMs, from eight popular vendors; Samsung, Xiaomi, ZTE, Nubia, Oppo, Amazon, Tecno, and Vivo. Column 1 in Table 6 details the device model and version for each ROM.

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

7.2 Impact of Target Selection Criteria

Column 9 shows the number of variables after applying the selection criteria §5.1, along with the percentage reduction (i.e., the number of reduced variables over the total number of recovered custom fields). As listed, the criteria lead to a substantial reduction of variables, averaging 94.5% for all analyzed ROMs.

False negatives A downside of the selection criteria is the possibility of false negatives. Given the absence of ground truth, we resorted to manual analysis to estimate FNs, carried out by two authors, who analyzed 5% of randomly-selected discarded variables (812 out of 17295 variables for AOSP and 1812 out of 38254 variables for custom ROMs) by checking whether they are connected to data holders via relations carrying AC implications (Table 1). We did not find any such cases. To estimate the maximum possible FN rate, we use Clopper-Pearson Exact Method [9] for binomial proportions with zero observed FNs. We calculate the confidence upper bounds using the formula $1-\alpha^{1/n}$, where n is number of variables (2624 in our case), and α (0.05 for a 95% confidence interval, and 0.01 for 99%) is the probability of a FN rate exceeding the computed upper-bound, leading to 0.1107% and 0.175% confidence upper bounds on the FN rate.

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

7.3 Inconsistency Detection Accuracy

Accuracy estimation. Due to the lack of ground truth for custom ROMs, we estimate the accuracy of Ariadne using AOSP (in line with prior work [11, 39, 44]). Specifically, we predict a normalized protection for each *target API* – i.e., those operating on data holders (§5.1), and compare it against its actual AC enforcement, implemented by the API. If the prediction is equal to the latter, we flag the case as correct.

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

Results. Column 4 in Table 6 reports the number of APIs that contain a data-holder-related ('data-related' for brevity) operation. These are the APIs covered by Ariadne. For custom ROMs, this number is for custom APIs. Overall, the ratio of data-related APIs differs across vendors, ranging from 11.8% in Oppo A16S and reaching up to 47.2% in Samsung A2058,

Table 6: ROMs Analysis Statistics

	1. ROM	2. APIs	3. Custom APIs	4. Covered APIs	5. Inconsistent APIs	6. Correct Predictions	7. Fields	8. Custom Fields	9. Analyzed Fields (% Reduction)	10. Edges Generated
AOSP	Pixel 5 (v12)	1513	-	326 (21.5%)	10	315 (96.6%)	17656	-	993 (94.3%)	1438
	Pixel 6 Pro (v13)	944	-	244 (25.8%)	5	238 (97.5%)	16935	-	1085 (93.5%)	1745
Custom	Xiaomi Mix 2S (v8)	2603	361	109 (30.1%)	3	-	27726	2747 (9.9%)	1736 (93.7%)	4041
	Amazon Fire HD (v10)	2896	821	228 (27.7%)	8	-	44603	16681 (37.4%)	2720 (93.9%)	4065
	ZTE Axon 40 (v12)	1995	514	115 (22.3%)	12	-	23988	6044 (26.1%)	1279 (94.6%)	2489
	Nubia Z50 Ultra (v13)	2076	482	107 (22.1%)	12	-	24410	7456 (30.5%)	1361 (94.4%)	2786
	Samsung A3058 (v10)	5888	2881	1361 (47.2%)	45	-	65160	26209 (40.2%)	3238 (95%)	10170
	Samsung Z Flip (v13)	5822	4218	1691 (40%)	35	-	65483	45027 (68.7%)	3083 (95.2%)	7776
	Samsung Tab S9+ (v14)	3767	2177	1203 (55.2%)	19	-	33934	14523 (42.7%)	1597 (95.2%)	11478
	Oppo A16S (v12)	1929	423	50 (11.8%)	2	-	22760	5538 (24.3%)	1179 (94.8%)	2085
	Oppo OP574FL1 (v14)	2010	512	86 (16.7%)	3	-	25414	6634 (26.1%)	1241 (95.1%)	5075
	Vivo V2309 (v14)	3507	849	167 (19.6%)	6	-	47485	13819 (29.2%)	2897 (93.8%)	6021
	Tecno Spark (v13)	3596	2002	720 (35.9%)	18	-	42985	19172 (44.6%)	2290 (94.6%)	16681

averaging 27.8% of the custom, and 23.6% of AOSP APIs.

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

AOSP False positives. We consider the cases where Ariadne projects an AC recommendation that is stronger than the implementation as false positives. As shown, our tool has 2.95% FP rate. We manually investigate the reported FPs and identify the following root causes:

Over-approximation of AC initial beliefs. To extract the initial beliefs, Ariadne relies on the approximation that an AC protects all data holder operations that *are control-dependent* on the AC check. This approximation may lead to inaccuracies as not all control-dependent operations are necessarily related to the AC (consider the case where an AC SYS_OR_SIG (Table 3) precedes a sensitive native method and an insensitive SET (Table 2) operation of var var_1, Ariadne will construct an inaccurate initial belief: [var_1,SET] requires AC SYS_OR_SIG). Poirot [11] can address this challenge by reasoning about protection targets. Hence, it is beyond the scope of this work. We discuss in §8 how an approach that combines both solutions (Poirot and Ariadne) can address this limitation effectively.

Inaccurate runtime type resolution. Resolving Java runtime types statically is a hard problem [5, 20, 31]. We use WALA to create context-sensitive callgraphs using 0-1CFA algorithm; however, runtime types of some variables cannot be determined accurately, which may affect the accuracy of the AC dependency graph construction.

7.4 Inconsistency Detection in Custom ROMs

We found that 17% of framework dex files (containing system service implementations) in the 11 custom ROMs cannot be correctly decompiled or processed by WALA (in particular, the Vivo ROM). Among the rest, Ariadne discovered a total of 163 data-driven inconsistencies, out of which 90 are unique. Column 5 in Table 6 shows a detailed breakdown. Every single ROM contains data-driven inconsistencies, ranging from two in Oppo A16S to 45 in Samsung ZFlip.

Due to the lack of ground truth in custom ROMs, we resort to manual analysis to evaluate the reported positives. Two authors inspected the decompiled code of each reported case and labeled it into one of three categories: (1) *true positive* – the API indeed lacks an AC enforcement along the path to a data holder and thus can be exploited, (2) *false positive* – the data holder accessed by the API is not sensitive and thus does not require AC, and (3) *unresolved positive* – it is unclear whether the API should enforce AC because of a proprietary feature, necessitating further examination by developers familiar with the codebase (i.e., vendor framework developers). Figure 4 shows a breakdown of positives: On average, 11.8% are FPs (with similar root causes as AOSP), 32.8% unresolved positives, and 55.1% TPs.

To demonstrate the security impact of TPs, we built end-to-end proof-of-concept exploits, listed in Table 7 with a range of security/privacy consequences: disabling highly-critical system packages such as the “android" package itself in Tecno, compromising a critical security policy in Samsung, launching random app Activities with system privilege in Vivo (i.e., an app can trigger protected Activities without requiring a permission), inferring currently running apps on the phone, and others. We have responsibly reported these to the vendors. At the time of writing, 11 are acknowledged, and two are under verification. We manually analyzed each and confirmed that seven are exclusively detected by Ariadne due to their

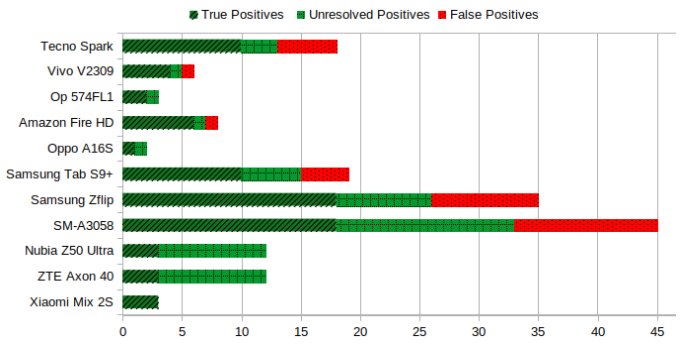

Figure 4: Categorization of Positives Reported by Ariadne

data-driven nature. We describe one case in detail in §9.

7.5 Availability and Impact of Edge Types

An essential requirement for the success of Ariadne is the availability of (deterministic and non-deterministic) edges in the AC dependency graph. Column 10 in Table 6 reports the total number of detected edges, which ranges from 1438 in AOSP up to 16681 in Tecno Spark. The custom ROMs (rows 3-13) introduce more services (on average 719 per ROM vs 378 in AOSP), APIs (up to 4218 in Samsung Z-FLIP), and data-holders (on average 2056 per ROM vs 1039 in AOSP), contributing to significantly more edges. We further study the relative impact of each edge type on AC predictions by examining its overall contribution to the total number of generated edges. The distribution follows a similar pattern across ROMs (§A.3) – where the similarity edges are the most common, followed by Class-1-Member. Other edges are less prevalent; nonetheless, helpful in detecting true inconsistencies (§9).

7.6 Impact of cut-off *relevance_weight*

Ariadne relies on a cut-off threshold to select high-confidence AC recommendations (§6.3). We evaluate the impact of this threshold on the overall results. We select three cut-off values: 0, 0.25 and 0.4, and calculate the total number of true positives, unresolved positives and false positives for Samsung ZFlip (Table 8). 0.25 represents the optimum value, providing a balance.We repeated this experiment for the rest ROMs, which revealed a similar trend. We omit the details for brevity.

7.7 Impact of Variations in *reduction_factor*

We selected *reduction_factor* values using domain knowledge (§6.2) to reflect our degree of confidence in each AC implication. A higher *reduction_factor* indicates stronger confidence (i.e., lower uncertainty), and vice-versa. To evaluate the sensitivity of Ariadne to variations in *reduction_factor*, we changed its value in four experiments:

Exp 1. Rule 1 from 0.95 to 0.9
Exp 2. Rule 1 from 0.95 to 0.8
Exp 3. Rule 5 from 0.6 to 0.65
Exp 4. Rule 7 from 0.4 to 0.2

Table 9: Impact of *reduction_factor* on inconsistencies

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

We counted the number of inconsistencies reported for two sample ROMs under the modified values (Table 9). The results show that changes in *reduction_factor* have a negligible impact on the reported inconsistencies. Only one (1/35) of the inconsistencies was missed in Samsung ZFlip. Our manual inspection showed that the affected case had limited data-driven AC implications, suggesting that the *reduction_factor* has a more pronounced effect when the number of edges is low.

8 Comparison with Prior Approaches

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

8.1 Convergence-based Approaches

Convergence-based approaches detect inconsistencies by comparing AC enforced along different paths accessing the same resource. Kratos [32], ACMiner [16] and AceDroid [2] look for inconsistencies within the framework, by identifying APIs converging on a shared resource.

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

To demonstrate this, we compare Ariadne against ACMiner [16], a state-of-the-art convergence-based inconsistency detection. We select ACMiner since its detailed inconsistency results are publicly available for a sample Android ROM (Nexus 5X – v7.1.1)[7]. We run Ariadne on the same ROM and compare our results to ACMiner's. We investigate Ariadne's FPs through manual investigation. ACMiner reports 28 (manually-confirmed) AC inconsistencies while Ariadne reports 26 inconsistencies, which partially overlap.

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

Table 7: Summary of Discovered Vulnerabilities

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

*The API is removed in the latest versions.

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

Table 8: Positive Reports for Samsung ZFlip

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

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

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

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

8.2 Probabilistic Approaches

Two probabilistic-inference approaches, Poirot [11] and Bluebird [39], have been proposed to alleviate some shortcomings of pure convergence-based tools. They both account for the uncertainty surrounding AC specification inference. Poirot suppresses FPs of convergence-based technique by pinpointing accurate targets of AC enforcement. It relies on *hints* such as control-dependencies, naming, and trigger-conditions. Bluebird [39] on the other hand, detects AC gaps in framework APIs by learning from system apps. It is limited in scope as it cannot tackle APIs not invoked by apps.

None of the prior tools models implicit AC relationships among Java objects or those arising from the type / granularity of underlying operations. Ariadne thus complements them by capturing a new class of AC inconsistencies.

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

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

Rediscoveries. Ariadne rediscovers 4 previously reported

cases by Poirot and/or Bluebird. E.g., `setScheduleType` (originally from Bluebird), and `setAmazonFlags` and `removeAmazonFlags` (originally from Poirot).

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

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

9 Case Studies

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

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

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

Listing 3: ZTEPowerTrackerService with two APIs

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

Listing 4: Data holders used by ZTEPowerTrackerService

10 Related Works

Framework-Vulnerability Analysis. Prior work for constructing Permission-to-API maps [6, 7, 10, 13] paved the path for Android vulnerability analysis.

Ariadne is closely related to inconsistency analyses that identify AC issues in APIs through convergence analysis or probabilistic inference. A comparison with ACMiner [16], Poirot [11], and Bluebird [39], three inconsistency detection tools, is detailed in §8. Kratos [32] and AceDroid [2] are similar to ACMiner – diferring in the granularity of recovered AC. Cross-layer inconsistency detection tools compare resources across layers like Native/Java [50] and Linux-file permissions/Java file access protections [18]. Dynamo [10] uses dynamic analysis to improve the results from static approaches. Other framework-layer vulnerability analyses are: ARF [17] identifies permission re-delegations, LeakDetector [49] intents that can leak sensitive data, and Cusper [38] and CuPerFuzzer [22] investigate custom permissions.

App-Vulnerability Analysis has also been the focus of prior works. IccTA [21] uses data flow analysis and symbolic execution, while AppIntent [45] uses taint analysis to identify data leaks. FIRMSCOPE [12] discovers privilege-escalation vulnerabilities in pre-installed apps. [8] analyzes the FOTA apps and identifies privacy and intrusiveness issues. Chex [23] identifies apps with components hijacking, whereas Activity-Hijacker [42] demonstrates how activity hijacking can lead to stealing sensitive user data. [29] and [48] analyze task and reference hijacking leading to ransomware and spyware.

Flow Networks and Flooding. Flow networks have been used to model and solve problems like network routing [26] and traffic management [34] to maximize the throughput or efficiency of the system [19]. Flooding algorithms are used for broadcast protocols [3], peer-to-peer networks [27], and sensor networks [46].

Flow networks have also been used in detecting anomalies [24], modeling attack surfaces [47], and mitigating DoS attacks [43]. Flooding techniques have been used in penetration testing [36] and preventing DoS attacks [41]. Ariadne uses flow networks to model AC annotations as flows and controlled flooding to model uncertain propagation (§6).

11 Ethics Consideration

In this section, we clearly explain the ethical considerations related to our research on finding Data-Driven Inconsistencies in Android.

We first identify the stakeholders who could be affected by this research. These include device vendors (such as Samsung, Tecno, Xiaomi, Vivo, and Amazon), Android users, and malicious actors. Device vendors might face harm if vulnerabilities in their devices are used by attackers before they are fixed. To manage this risk, we responsibly shared all vulnerabilities we found with the respective vendors ahead of time. We gave vendors detailed POCs and ample time to fix the problems before the potential publication of our findings. We will not publicly disclose details of vulnerabilities that are not fixed.

Android users could experience privacy issues or unauthorized access to their devices if data-driven vulnerabilities are exploited. Our tool can help vendors detect and fix such vulnerabilities, thereby ensuring users data privacy and security. We note that such risk is not applicable in the specific evaluation-scenarios of our research because we performed PoC testing on dedicated test devices and did not involve any real user devices.

There is also a risk to the broader community because malicious actors might misuse our methods for finding security vulnerabilities. Since our tool is open-source, we encourage responsible use and emphasize the importance of ethical practices when using our methods.

During the research, we did not encounter any unexpected ethical problems. However, we recognize that better cooperation with vendors could have led to quicker acknowledgement and fixes of reported vulnerabilities. Future work could benefit from closer partnerships with vendors to speed up their responses.

12 Open Science

We fully open source Ariadne's source code and dataset at https://doi.org/10.5281/zenodo.15612788.

Acknowledgments

This research was supported, in part by NSERC under grant RGPIN-07017, by the Canada Foundation for Innovation under project 40236, by Intel Labs, and by a Google ASPIRE award. This work benefited 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] Android dumps, 2024. https://dumps.tadiphone.dev/dumps.

[2] Yousra Aafer, JianJun Huang, Yi Sun, Xiangyu Zhang 0001, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[3] Syaiful Ahdan, Hamonangan Situmorang, and Nana Rachmana Syambas. Effect of overhead flooding on ndn forwarding strategies based on broadcast approach. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–4. IEEE, 2017.

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

[5] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 794–807, New York, NY, USA, 2020. Association for Computing Machinery.

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

[7] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Revisiting android permission specification analysis. In *25th {USENIX} security symposium ({USENIX} security 16)*, pages 1101–1118, 2016.

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

[9] C. J. CLOPPER and E. S. PEARSON. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 12 1934.

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

[11] Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. Poirot: Probabilistically recommending protections for the android framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 937–950, New York, NY, USA, 2022. Association for Computing Machinery.

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

[13] Adrienne Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. pages 627–638, 10 2011.

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

[15] Rossano Gaeta and Matteo Sereno. Generalized probabilistic flooding in unstructured peer-to-peer networks. *IEEE transactions on parallel and distributed systems*, 22(12):2055–2062, 2011.

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

[17] Sigmund Albert Gorski and William Enck. Arf: Identifying re-delegation vulnerabilities in android system services. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 151–161, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. FReD: Identifying file Re-Delegation in android system services. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1525–1542, Boston, MA, August 2022. USENIX Association.

[19] Zhipeng Jiang, Xiaodong Hu, and Suixiang Gao. A parallel ford-fulkerson algorithm for maximum flow problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 70. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2013.

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

[21] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

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

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

[24] Huy Anh Nguyen, Tam Van Nguyen, Dong Il Kim, and Deokjai Choi. Network traffic anomalies detection and identification with flow monitoring. In *2008 5th IFIP International Conference on Wireless and Optical Communications Networks (WOCN'08)*, pages 1–5. IEEE, 2008.

[25] Adamou Ouorou, Philippe Mahey, and J-Ph Vial. A survey of algorithms for convex multicommodity flow problems. *Management science*, 46(1):126–147, 2000.

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

[27] Marius Portmann and Aruna Seneviratne. Cost-effective broadcast for fully decentralized peer-to-peer networks. *Computer Communications*, 26(11):1159–1167, 2003.

[28] Ashikur Rahman, Wlodek Olesinski, and Pawel Gburzynski. Controlled flooding in wireless ad-hoc networks. In *International Workshop on Wireless Ad-Hoc Networks, 2004.*, pages 73–78. IEEE, 2004.

[29] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 945–959, 2015.

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

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

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

[33] 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.

[34] Banavar Sridhar, Shon R Grabbe, and Avijit Mukherjee. Modeling and optimization in traffic flow management. *Proceedings of the IEEE*, 96(12):2060–2080, 2008.

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

[36] Deris Stiawan, Mohd Yazid Idris, and Abdul Hanan Abdullah. Penetration testing and network auditing: Linux. *Journal of information processing systems*, 11(1):104–115, 2015.

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

[38] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A Gunter. Resolving the predicament of android custom permissions. In *NDSS*, 2018.

[39] Parjanya Vyas, Asim Waheed, Yousra Aafer, and N. Asokan. Auditing framework APIs via inferred app-side security specifications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6061–6077, Anaheim, CA, August 2023. USENIX Association.

[40] WALA. Wala, Accessed on: November 27, 2024. https://github.com/wala/WALA.

[41] Haopei Wang, Lei Xu, and Guofei Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 239–250. IEEE, 2015.

[42] Zhaoguo Wang, Chenglong Li, Yi Guan, Yibo Xue, and Yingfei Dong. Activityhijacker: Hijacking the android activity component for sensitive data. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2016.

[43] Anthony D Wood and John A Stankovic. Denial of service in sensor networks. *computer*, 35(10):54–62, 2002.

[44] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634, 2013.

[45] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054, 2013.

[46] Kasim Sinan Yildirim and Aylin Kantarci. Time synchronization based on slow-flooding in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):244–253, 2013.

[47] Changhoon Yoon, Seungsoo Lee, Heedo Kang, Taejune Park, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Flow wars: Systemizing the attack surface and defenses in software-defined networks. *IEEE/ACM Transactions on Networking*, 25(6):3514–3530, 2017.

[48] Wei You, Bin Liang, Wenchang Shi, Shuyang Zhu, Peng Wang, Sikefu Xie, and Xiangyu Zhang. Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices. In *Proceedings of the 38th International Conference on Software Engineering*, pages 959–970, 2016.

[49] Hao Zhou, Xiapu Luo, Haoyu Wang, and Haipeng Cai. Uncovering intent based leak of sensitive data in android framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3239–3252, New York, NY, USA, 2022. Association for Computing Machinery.

[50] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. 2022.

A Appendix

A.1 Similarity between two nodes of the AC Dependency Graph

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

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

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

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

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

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

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

A.2 Additional Case Studies

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

A.3 Edge Types

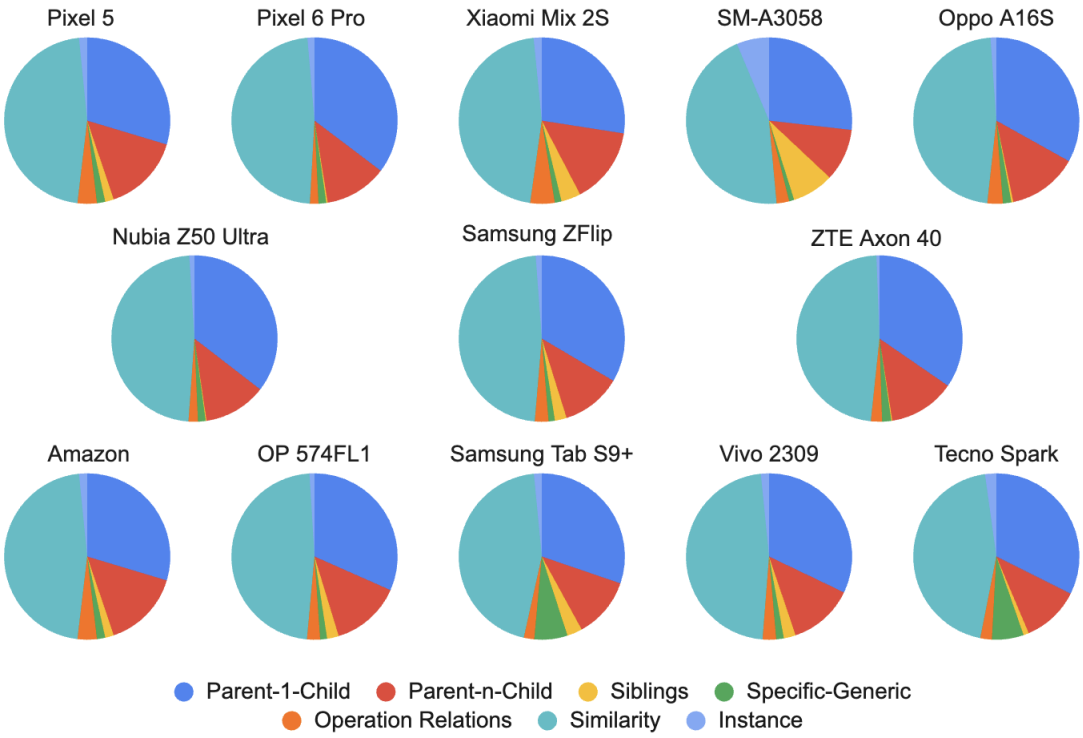

Figure 5: Distribution of Edge Labels