Auditing Framework APIs via Inferred App-side Security Specifications

Parjanya Vyas, Asim Waheed, Yousra Aafer, and N. Asokan, University of Waterloo

https://www.usenix.org/conference/usenixsecurity23/presentation/vyas

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.
August 9–11, 2023 • Anaheim, CA, USA
978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.
Auditing Framework APIs via Inferred App-side Security Specifications

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

Asim Waheed  
University of Waterloo  
asim.waheed@uwaterloo.ca

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

N. Asokan  
University of Waterloo  
asokan@acm.org

Abstract

In this work, we explore auditing access control implementations of Android private framework APIs by leveraging app-side security specifications. The seemingly straightforward auditing task faces significant challenges. It requires extracting unconventional security indicators and understanding their relevance to private framework APIs. More importantly, addressing these challenges requires relying on uncertain hints. We hence, introduce Bluebird, a security auditing platform for Android APIs, that mimics a human expert. Bluebird seamlessly fuses human-like understanding of app-side logic with statically-derived program semantics using probabilistic inference to detect access control gaps in private APIs.

1 Introduction

Android device manufacturers often define private APIs in the framework, which are typically invoked by preloaded apps to access custom functionality. Ensuring complete mediation along the path from a preloaded app entry point to a private API’s functionality is a security-critical process that requires careful coordination of security specifications across the app and framework layers. Both parties, preloaded app developers and framework developers, should enforce similar security measures. This is a challenging task, particularly due to the lack of security specifications for private APIs. Preloaded apps may fail to protect access to an invoked private API in face of external requests, thus introducing hijack-enabling flows. An unprivileged (malicious) app can leverage such flows as a stepping stone to trigger the private API, thus sidestepping potential framework-side access control. This classic scenario – reflecting one type of complete mediation lapse, has been widely studied in the Android literature [23].

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

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

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

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

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

Addressing these challenges is a highly uncertain process, as we cannot precisely deduce that a given (UI-based) app entry is sensitive, nor can we derive the exact target of an app entry’s access control. Rather, we can mimic (uncertain) human-like reasoning by relying on various hints. For example, a human security analyst can speculate that an app dialog

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

2 An app component or a UI block.
AOSP APIs are found to contain no gaps, based on their work. Next, (e.g., using NLP techniques) with statically-derived app-side versions 8.1, 9.0, 10, 11, 12 and 13), and 7 Android custom ROMs. Our evaluation on the custom ROMs, from Samsung, Amazon, Oppo, Vivo, Xiaomi, and ZTE. Our evaluation on the AOSP codebases, demonstrates concrete access control check.

A gap a perspective. Note that the API’s access control if the posterior marginal probability does not indicate the actual missing access control, but rather indicates the presence of a gap, since it is difficult to map the inferred sensitivity to a concrete access control check.

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

We claim the following contributions:

- We propose Bluebird, an automated probabilistic security audit technique for unprotected Android APIs that is capable of leveraging and fusing uncertain app-side security logic. (Section 3)
- We develop a set of probabilistic inference rules that allow pairing human-like understanding of app-side security logic with statically derived program semantic and artifacts to conduct the security audit procedure. (Section 4)
- Via extensive empirical evaluation on 14 different ROMs (Section 5), we demonstrate the effectiveness of Bluebird by identifying 11 specific vulnerabilities (Table 5), all of which have been acknowledged by the vendors involved, and 6 have been granted bug bounties. (Sections 5, 6)

2 Motivation

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

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

In the second scenario (Figure 1), the API is triggered upon launching the Activity FolderContainer. Besides the user consent check at Line 5, there is a more-difficult to detect access control: The user needs to enter her password in the Fragment (Figure 1.A.), and click on Continue button before landing on the FolderContainer Activity. Observe that although FolderContainer is exported (Figure 1.C), the access control check (Line 5 in Figure 1.D) implies that the user needs to be logged in as user id >= 150.
enableBluetooth(); ...
unInstallDisallowedApps();
setSecureFolderPolicy("DisallowPackage", mDisallowList, userId);
}

We can confirm that our earlier deduction – based on the (simplified) implementation depicted in Listing 1 (extracted from Samsung SM-A3058):—

```
Listing 1: PolicyUpdateReceiver Manifest Definition

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

Listing 2: PolicyUpdateReceiver Code Listing

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

```
Listing 3: setSecureFolderPolicy

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

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

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

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

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

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

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

Consider the code listing of PolicyUpdateReceiver (Listing 2). We can deduce that the following are main contributors:

1. The internal method `downloadPolicy()` is a likely main contributor to the Receiver’s functionality. It shares the common word `policy` with the Receiver’s name.
2. `setSecureFolderPolicy` (line 11) is a likely main contributor to the private method `downloadPolicy()` as they share a common word `policy`. Give our observation in (1), we can transitively propagate our conclusion up to the Receiver: `setSecureFolderPolicy` is a likely main contributor to the Receiver’s functionality.
3. We can reinforce our transitive conclusion in (2) by another observation. `setSecureFolderPolicy` and the Receiver’s required permission (`RECEIVE_SECURE_FOLDER_POLICY_UPDATE`) (Listing 1) share a stronger naming similarity. Thus, we are more confident now that `setSecureFolderPolicy` is a likely main contributor to the Receiver’s functionality.
4. The sdk API `enableBluetooth` and the private method `uninstallDisallowedApps` are likely main contribu-
tors to the calling method `downloadPolicy`. Although we cannot spot a syntactic naming similarity, we know that the words `enable` and `disallow` are related to the concept of `policy`. We can use similar transitive reasoning as in (2) to deduce that `enableBluetooth` and `uninstallDisallowedApps` are likely main contributors to the Receiver.

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

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

**Reasoning scope:** We limit our reasoning scope to code elements that are (directly or transitively) related to target APIs via call dependencies; that is, we only estimate the contribution scope of target private APIs and their (transitive) callers. This focused reasoning is driven by our domain-based hint that analyzing code unrelated to the APIs cannot enhance our belief in the API’s contribution extent. For example, although we can conclude that the internal method `setFolderContainer` (at line 6, Figure 1.D) is a main contributor to `FolderContainer` Activity, it does not impact our conclusion regarding the contribution of the target private APIs (because a component may have many main contributors).

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

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

We identified two main reasons leading to these cases:

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

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

This category of versatile APIs can impede our proposed auditing. Without resolving the context, a naive auditing tech-

---

\(^3\) Apps or the user.
nique may associate a seemingly relaxed access control with a potentially sensitive API. Consider the invocation site of `setSecureFolderPolicy` in Listing 2, line 11. The third argument `userId` constrains the policy change to the scope of the calling user (i.e., `getCallingCurrentUserId`); that is, a user can only change her own policy. The argument hence can be interpreted as an implicit security property, which should likely be enforced by the API implementation (e.g., by verifying if the argument matches a specific value).

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

Our Solution. Inspired by the recent application of probabilistic inference in Android security [11], we resort to modeling the audit task as a probabilistic inference problem. Probabilistic inference allows drawing conclusions from uncertain information and observations by modeling them in the form of a probability distribution. Specifically, we need to compute the marginal probability that an API is sensitive from various app-side hints. The probability depends on the sensitivity of the invoking app entries, and is conditional on various properties connecting the API to the entries.

In probabilistic inference, a random variable (a boolean variable with a probability) is used to denote a predicate (e.g., “an API is sensitive” or “an app entry is sensitive”). Interdependent predicates (e.g., “an API is likely sensitive if it is a main contributor to a sensitive app entry”) are represented as probabilistic implication constraints, as follows: \( \text{pred}_1 \rightarrow \text{pred}_2 \) where \( \text{pr} \) denotes a (prior or computed) confidence in \( \text{pred}_1 \) implying \( \text{pred}_2 \) to be true. The predicates may be involved in multiple constraints denoting their dependencies. Probabilistic inference aggregates the constraints, which are considered together to form a joint distribution, and computes a posterior marginal probability of the APIs’ sensitivity. Intrinsically, the final probability reflects a better sensitivity estimation, as the impact of uncertainties and contradictions will be overpowered by dominant hints. Next, we discuss our technique in more details, and explain how it can be used to detect access control gaps.

3 Our Technique

Figure 2 presents a high level overview of our probabilistic security audit system, Bluebird. It consists of three phases: Phase 1 - framework-side analysis to determine priors: Bluebird begins by statically analyzing the framework-side to identify Android APIs and pertaining access control logic.

For each framework API, Bluebird introduces a random variable that denotes the probability of the API being sensitive. The random variable is initially designated a sensitivity level, called prior-probability, which has two possible constant values: \( p_h \) denotes that the API is likely sensitive (because static analysis reveals that the API enforces access control), and \( p_i \) denotes that it is not.

Existing literature [9, 19, 32] of probability inference leverage predefined prior probability values derived from domain knowledge and show that inference results are typically insensitive to these initial values. Here, we follow the same practice, by setting 0.9 for \( p_h \) and 0.1 for \( p_i \).

Given our goal, Bluebird limits the audit procedure to APIs with an initial \( p_i \) prior-probability.

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

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

Second, Bluebird moves to collecting clues. Unlike evidence, clues are less certain human-reasoning rules that guide the inference procedure to derive a sensitivity level for an API based on its invocation in apps. Specifically, Bluebird looks for two kinds of clues, contribution extent and contextual features, which are designed to solve the challenges discussed in Section 2. Clues are conditional and are regulated by an implication probability; that is, if a property holds, Bluebird propagates an app-side sensitivity to invoked APIs with some...
probability. Note that the probability reflects the uncertain nature of the clues. For example, we can only state that An API is likely sensitive because it is highly relevant to the sensitivity indicator in an app entry with a (computed) confidence.

**Phase 3: fuse priors with probabilistic constraints to compute posterior probabilities:** Bluebird leverages a probabilistic inference algorithm [21] to fuse the priors and probabilistic constraints to compute marginal posterior probabilities of individual APIs being sensitive.

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

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

Next, Bluebird analyzes the preloaded app SecureFolder (i.e., the invoking app), to derive a sensitivity-level for each app entry leading to the API. Based on traditional access control analysis, the Receiver PolicyUpdateReceiver (Listings 1 and 2) is assigned a high sensitivity score \( p_h \) since it uses a system-level permission.

Based on NLP, UI and component chaining analysis (details are explained later in Section 4), the Activity FolderContainer is assigned a computed sensitivity score 0.99. Note that the high score is deduced from two evidences: (1) the physical user id (Line 5 in Figure 1.D) and (2) the Activity is reachable from a highly sensitive fragment (Figure 1.A).

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

**Contribution Extent.** Bluebird constructs contribution probabilistic constraints to encode the speculation that an API is likely relevant to the sensitivity indicators in the app with a probability relative to its contribution extent. Accordingly, Bluebird proceeds by mimicking the reasoning of a human analyst as in Section 2 to estimate a contribution score for setSecureFolderPolicy to the Receiver – which yields a score = 0.75 (details explained later in Section 4). It then propagates the Receiver’s sensitivity \( p_h \) to setSecureFolderPolicy with the implication probability: \([p_h \times \text{Contribution score}] = 0.71\). We use the probability factor \( p_h \) to account for the uncertainty that developers may not always obey the naming conventions.

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

**Context Dependency.** Bluebird constructs context constraints to account for contextual properties that may influence the propagation of the priors. For the motivating example, Bluebird recovers the following property: setSecureFolderPolicy takes a special type of input (i.e., getCallingUserId) at the two app entry points. As such, it formulates two probabilistic constraints, each propagating the entry’s sensitivity to the API with high confidence (for a more elaborate explanation, refer to Section 4). Last, the inference procedure combines the priors and context constraints to project a final posterior marginal probability. The inference yields a final posterior \( p = 0.94 \) implying a gap in the API’s access control.

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

## 4 System Design

In this section, we dive into the details of Bluebird. We introduce the inference rules used by Bluebird in Table 1. The Table lists the predicates relevant to the rules, their symbols, definitions, and pertaining constraints.

### 4.1 Framework Analysis: Computing Priors

Bluebird starts by extracting framework-side security checks to estimate priors (i.e., initial sensitivity of APIs). For each API defined in the Android system services, it builds an interprocedural control flow graph and traverses it in a depth-first fashion to identify access control checks. It extracts permission, user, and process id checks by looking for corresponding enforcement methods, resolving relevant conditional statements, and tracking data flows to resolve arguments, if any. The analysis then normalizes the checks according to the scheme proposed by AceDroid [1]. Bluebird establishes the prior probability using Rule 1: an API \( api \) has \( p_i \) prior-probability of being sensitive if its framework implementation reveals a normalized access control corresponding at most to Normal level or to 06.

### 4.2 App Analysis: Extracting Sensitivity-Relevant Evidences

Bluebird begins app analysis by identifying direct app-side entries leading to (private) APIs through reachability analysis. Here, we rely on an existing tool, EdgeMiner [7] to connect implicit control flows. The reachability analysis yields a mapping of app entries and the APIs they (eventually) invoke. We note that ensuring a comprehensive mapping is challenging.

---

6AceDroid uses standard permission protection levels for normalization.
Table 1: Predicate/random variable and constraint definition

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Symbol</th>
<th>Definition</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitive</td>
<td>( S(ep, api) )</td>
<td>Entry point ( ep ) or API ( api ) is sensitive</td>
<td>( S(api) = 1(p_m) )</td>
</tr>
<tr>
<td>( F(api) )</td>
<td>Framework API ( api ) enforces weak or no access control</td>
<td>( S(api) = 1(p_m) )</td>
<td></td>
</tr>
<tr>
<td>( F(ep) )</td>
<td>App entry ( ep ) enforces weak or no access control</td>
<td>( S(ep) = 1(p_m) )</td>
<td></td>
</tr>
<tr>
<td>( ¬F(ep) )</td>
<td>App entry ( ep ) enforces Strong access control</td>
<td>( S(ep) = 1(p_m) )</td>
<td></td>
</tr>
<tr>
<td>( U(ep) )</td>
<td>App entry ( ep ) corresponds to a sensitive UI type</td>
<td>( U(ep) = 1(p_m) )</td>
<td></td>
</tr>
<tr>
<td>( D(ep) )</td>
<td>App entry ( ep ) is a non-disposable UI block</td>
<td>( D(ep) = 1(p_m) )</td>
<td></td>
</tr>
<tr>
<td>( C(ep) )</td>
<td>Entry ( ep ) enforces a double confirmation (UI)</td>
<td>( C(ep) = 1(p_m) )</td>
<td></td>
</tr>
<tr>
<td>( N(ep) )</td>
<td>App entry ( ep ) likely contains sensitive keywords</td>
<td>( N(ep) = 1(score) )</td>
<td></td>
</tr>
<tr>
<td>( T(ep, api) )</td>
<td>API ( api ) is a main contributor to ( ep ) based on NLP.</td>
<td>( T(ep, api) ) is the computed Contribution as explained in Section 4.3.</td>
<td></td>
</tr>
<tr>
<td>( M(ep, api) )</td>
<td>( api )’s context is fully modifiable through entry ( ep )</td>
<td>( M(ep, api) ) is the computed Modifiable context as explained in Section 4.3.</td>
<td></td>
</tr>
<tr>
<td>( ¬M(ep, api) )</td>
<td>The context of ( api ) is non-modifiable through entry ( ep )</td>
<td>( ¬M(ep, api) ) is the computed Non-modifiable context as explained in Section 4.3.</td>
<td></td>
</tr>
<tr>
<td>( UM(ep, api) )</td>
<td>( api )’s context is modifiable only through user input to ( ep )</td>
<td>( UM(ep, api) ) is the computed User-modifiable context as explained in Section 4.3.</td>
<td></td>
</tr>
<tr>
<td>( P(ep, api) )</td>
<td>API ( api ) takes an input hinting implicit ( AC ) in ( ep )</td>
<td>( P(ep, api) ) is the computed Special Input as explained in Section 4.3.</td>
<td></td>
</tr>
</tbody>
</table>

\( p_m \) is explained in Section 4.2.  
\( \dagger \) CT refers to the computed Contribution as explained in Section 4.3.

particularly for heavily obfuscated apps. A discussion of how obfuscation affects our analysis can be found in Section 7.

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

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

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

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

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

- If a UI-block type corresponds to known sensitive types (e.g., AlertDialog, android:inputType= textPassword), we associate the UI-block with a high sensitivity level with a medium confidence \( p_m \) – Rule \( 4 \).
- If a field attribute corresponds to a non-disposable type (i.e., cannot be dismissed), we similarly associate the UI-block with a high sensitivity with medium confidence – Rule \( 5 \).
- If the path leading to the invocation of the API from the UI-block contains more than one known UI based callbacks (e.g., multiple onClk), we associate the UI-block with a high sensitivity level with medium confidence – Rule \( 6 \).

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

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

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

Since our training corpus is relatively small, we use transfer learning to build the model. We start with a pre-trained BERT Cased model [12] and train it on our labeled data. This choice channels the knowledge learned by the pre-trained model to our target Android-specific domain. We used 1024 epochs with all layers unfrozen, and non-overlapping subsets of 20% data for validation and testing. Given the probabilistic nature of our analysis, we apply the softmax operation as the final scoring component and permission names, UI textual elements) and invoked methods (e.g., class, name, and non-primitive argument types). It then uses a transformer-based encoder to produce an embedding vector for the collected information. The contribution of a method to an entry point is estimated by the cosine-similarity of the produced embedding.

The following Equation details the contribution estimation.

\[
\text{Contribution}(ep, api) = \text{Max}(\text{Sim}(ep, api), \\
\text{Contribution}(ep, c(api)) \times \text{Sim}(c(api), api))
\]

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

### 4.4 App Analysis: Extracting Context Clues

Our idea here is to propagate the sensitivity of an app entry to the APIs it invokes while considering contextual properties. Rules [7] [12] depict the program artifacts that we rely on to generate the context constraints.

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

```
Listing 4: API with path-sensitive Access Control
1 public void incrementOperationCount(int uid, ...) {
2   if (Binder.getCallingUid() == uid // Path 1
3     || mContext.checkCallingPermission(MODIFY_NETWORK_ACCOUNTING, ...) == 
4     1 // Path 2
```

We note that this conditional nature complicates our auditing task – without reasoning about the execution context, we might incorrectly propagate the app’s specification to called APIs. For example, consider the simplified access control implementation of the AOSP API NetworkService.incrementOperationCount, shown in Listing 4. The API features two distinct access control paths. The first path
requires the supplied \textit{uid} to be equivalent to the calling app \textit{uid}, while the second one requires the calling app to hold a specific permission.

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

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

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

\textbf{Data Sensitivity.} App entries invoking \textit{data-sensitive} APIs are likely to impose a protection dependent on data flows. Unfortunately, due to the lack of a consensus or a guidance of what constitutes a \textit{data-sensitive} API, particularly in light of customization, it is challenging to identify whether an API is data-sensitive, at least without extensive framework analysis\footnote{SuSi \cite{3}'s classification is not fully applicable as it is mostly concerned with detecting sinks from the perspective of a remote adversary.}

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

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

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

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

\section{4.5 Probabilistic Constraint Solving}

The inference constraints can be represented as probabilistic functions over the random variables involved. In practice, the computation of posterior marginal probability over all these constraints is expensive, particularly in the cases of a large number of constraints. In our implementation, we leverage a graphical model, factor graph \cite{21}, to represent all probabilistic functions and allow an efficient computation. We use a standard off-the-shelf algorithm to compute posterior probabilities for the factor graphs. Our implementation is based on \textsc{ProbLog} \cite{22}, an off-the-shelf factor graph engine (The details are hence elided). Refer to Appendix A.2 for more details about the \textsc{ProbLog} rules corresponding to Table 1.

\section{4.6 Interpreting Auditing Results.}

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

Bluebird can lead to the three possible outcomes listed in Table 2. Category 3 reflects access control gaps, that is cases that indicate potential security issues and thus warrant further investigation. (Note that these outcomes do not reflect the obvious scenario where our audit cannot intrinsically work; i.e., where there are no apps invoking an API. We report the percentage of such cases in Section 5.2).

\section{5 Evaluation}

We implemented a prototype of Bluebird\footnote{We will make the source code for research use available on request.}, which comprises three modules as illustrated in Figure 2. The modules are responsible for: (1) static framework analysis, (2) static/UI app analysis, and (3) probabilistic inference. The static analysis
modules are built on top of WALA [29] and analyze framework / preloaded app bytecode to construct analysis rules (priors and constraints) required for the inference. The probabilistic inference module fuses and solves the constraints.

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

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

### 5.1 Test ROMs

To test our Bluebird prototype, we collected a total of 14 ROMs, from public online repositories [14, 15] and from available physical devices. The samples include both AOSP and custom Android ROMs and span six versions, Android 8.1, 9, 10, 11, 12, and 13. They are customized by six vendors, at both ends of the customization spectrum.

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Initial Sensitivity</th>
<th>Final Sensitivity</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HIGH</td>
<td>-</td>
<td>Do not Audit: No security issue</td>
</tr>
<tr>
<td>2</td>
<td>LOW</td>
<td>LOW</td>
<td>Compliance: No security issue</td>
</tr>
<tr>
<td>3</td>
<td>LOW</td>
<td>HIGH</td>
<td>Access Control Gap: Likely security issue</td>
</tr>
</tbody>
</table>

Bluebird specifically targets private vendor APIs; nonetheless, we report the results for public APIs for comparison.

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

### 5.2 Availability of App-Side Security Specifications

Our proposed approach hinges on the availability of invocation sites of APIs in apps. Besides, given the uncertainties involved in (1) connecting APIs to the app-side security specifications, and in (2) even possible contradictions in inferred API security specifications, our approach would work best when multiple invocation sites are found for an API, as app-side security specifications would lead to substantial uncertainty when only a small number of invocations are involved.

We thus report the number of invoked public APIs, invoked private APIs, and the average number of invocation sites per API in preloaded apps, in Columns 5, 6 and 8, respectively.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Invoked public APIs</td>
</tr>
<tr>
<td>6</td>
<td>Invoked private APIs</td>
</tr>
<tr>
<td>8</td>
<td>Average number of invocation sites per API</td>
</tr>
</tbody>
</table>

### 5.3 Evaluating Auditing Results on AOSP

For a perfectly secure ROM – where APIs are protected consistently across the app and framework layers, Bluebird should report results belonging to category 2, as long as there are sufficient evidences and clues. Hence, we rely on this intuition to gauge the accuracy of our tool. As in prior work [11], we assume that AOSP is perfectly secure, and use it to evaluate Bluebird. Column 7, Table 3 reports the number of public APIs that Bluebird identified as having access control gaps in AOSP. As shown, Bluebird reports a gap for 8% to 12% APIs which are by our assumption false positives. This means that the remaining public APIs (88% to 92%) were found to be consistent with their app-side security specifications.

We investigated all reported FPs manually – two authors were involved. The investigation identified that FPs were caused by the following: (1) limitation of our access control extraction logic in the framework-side (i.e., the API uses a difficult to detect logic), (2) limitations of app-side access control logic due to disconnections in the ICFG, (3) insufficient app-side evidences, and (4) apps being overprotective. We did not attempt to verify whether they are indeed FPs since we consider AOSP as ground truth.

Observe that quantifying false negatives (FNs) is challenging due to the lack of ground truth. We resort to approximating FNs by constructing a synthetic vulnerable APIs dataset – where we maneuver Bluebird to assign a LOW prior to AOSP APIs that implement strong access control (i.e., we intentionally change the prior from HIGH to LOW). Refer to Appendix A.3 for a sample of APIs used to create the synthetic dataset. We then run Bluebird and count the portion of cases where it reports a LOW posterior, indicating an FN. On average, our analysis yields 17% FN rate. We manually investigated the cases and found that they are mostly due to: (1) An inferred LOW indicator for a sensitive UI blocks, and (2) insufficient app-side evidences.

### 5.4 Landscape of Gaps in Custom ROMs

We then run Bluebird on custom Android ROMs. Column 7 in Table 3 reports the number of private APIs with access control gaps. As shown, the number of gaps ranges from 15% to 44%. Due to the lack of ground truth, we estimate false positives in the reported gaps through manual investigation. Two...

---

11Category 1 APIs are excluded as they implement strong access control.
authors examined 10% of the gaps and an FP is reported if both agreed. The authors rely on domain knowledge to verify whether a gap is an FP. Examples of manually detected FPs are WifiManager.getWifiApInterfaceName() and Knox-
CustomManager.getVibrationIntensity. The FPs were caused by the same reasons as in AOSP analysis.

### 5.5 Impact of the Preset Probabilities

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

**EXP 1.** \( p_l = 0.35 \) and \( p_h = 0.8 \) in Rules 2 and 3.

**EXP 2.** \( p_h = 0.8 \) in Rules 7 and 8.

**EXP 3.** \( p_l = 0.2 \) and \( p_h = 0.8 \) in Rule 10.

**EXP 4.** \( p_m = 0.6 \) in Rule 11.

All experiments are performed in isolation (i.e., probabilities not mentioned in the experiment remain unchanged as in Table 1) and on the same ROM (Pixel 6 Android 11) to assess the impact of each change precisely and consistently.

Table 4 shows the number of APIs reported by Bluebird for each category using the four experimental settings. As shown, the impact of the probability values is largely minimal. We manually investigated the few API instances that did exhibit some change, and concluded it mainly occurred due to insufficient app-side clues/evidences – i.e., the preset probabilities may have higher impacts in cases where there are fewer hints.

### 5.6 Comparison with Convergence-Based Inconsistency Analysis

A closely-related line of research to our approach is convergence-based inconsistency analysis, which aims to identify access control flaws by comparing security specifications leading to similar resources. The works perform convergence analysis either within the framework layer [1, 16, 27] or across different layers of the Android stack [18, 36]. Bluebird is conceptually a cross-layer access control analysis framework, as it conducts the auditing procedure by learning security specifications from the app layer.

To qualitatively demonstrate the benefits of using Bluebird over existing convergence-based work, we investigate a subset of reported access control gaps. Particularly, we intentionally select those leading to actual vulnerabilities (Please refer to Section 6 for more details about the cases) and check whether prior works are able to detect them. Our analysis confirms that out of the 11 reported vulnerabilities, \textit{7 cannot be caught by existing convergence-based techniques}, as (1) the vulnerable APIs do not share a convergence point with other framework APIs, and (2) the vulnerable APIs do not contain any cross-layer access (e.g., no file access, and no native method invocation). A prominent example of these vulnerabilities is the case discussed in Section 2.

While this finding clearly demonstrates its complementary nature, Bluebird is inherently more coarse-grained compared to existing approaches. It can only indicate the presence of a gap (i.e., an access control is absent), while other work can

---

### Table 4: Impact of probability values

<table>
<thead>
<tr>
<th>Category</th>
<th>EXP1</th>
<th>EXP2</th>
<th>EXP3</th>
<th>EXP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>234</td>
<td>234</td>
<td>234</td>
<td>234</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>74</td>
<td>75</td>
<td>74</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>22</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>
identify precisely the missing access control (e.g., exact permission). As Bluebird relies on app-side sensitivity indicators, including natural language text from UIs, it cannot map those to concrete traditional access control checks.

5.7 Comparison with Poirot

Our work is inspired by Poirot [11], an Android access control analysis and recommendation tool, that similarly leverages probabilistic inference to account for inherent access control-related uncertainty. Contrary to our work, Poirot is specifically tailored towards framework-level access control analysis and works exclusively by relying on framework-level security-relevant hints. Bluebird is a parallel probabilistic security analysis framework for modeling of evaluating access control across the app and framework layers.

Here, we demonstrate the fundamental differences with Poirot. We obtained a list of access control inconsistencies reported by Poirot for Amazon Fire HD 10 and compared it against the results we obtained using Bluebird (refer to Row 8 in Table 3). Out of the 24 gaps reported by Bluebird, Poirot was able to identify three, hence missing 21. Besides, out of the 12 true positives reported by Poirot, Bluebird was only able to identify three. We present a sample case-study from each category with a brief root cause discussion to showcase the complementary nature of Bluebird and Poirot.

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

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

6 Case Studies of Access Control Gaps

Not all reported APIs with access control gaps are exploitable. Nonetheless, they do warrant further investigation as Bluebird probabilistically detects the gaps based on dominant app-side hints. To demonstrate that some gaps are indeed exploitable, we selected a few whose logic is comprehensible (some reports occur in custom proprietary functionality) and built end-to-end PoCs. Table 5 reports our manually confirmed vulnerabilities; which can allow third-party apps to alter various system settings and mount DoS attacks without permission. We reported the findings to the vendors. All reports have been acknowledged, and nine have been fixed. Next, we describe two selected cases. (Refer to Appendix for a discussion of three other cases).

Manipulating Samsung’s app security policies.
Bluebird reported a gap in PersonaManagerService.setAppSeparationDefaultPolicy, a private API in Samsung, that allows to reset app-specific separation policies. As policy updates can only be initiated by device administrators on Samsung, the API is clearly sensitive. However, we found that the API enforces no access control. Bluebird was able to identify the gap thanks to various UI-based evidences and clues in a preloaded app SecAppSeparation. We explain two examples next.

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

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

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

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

7 Threats to Validity and Limitations

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

Bluebird is dependent on the availability of app-side invocations of private APIs, and the availability of (detectable) sensitivity indicators, which poses a threat to the validity of our findings in Section 5. The number of private APIs’ invocations in the preloaded apps, analyzed in our collected ROMs corpus, may not be representative of other custom ROMs.
Existing research [10] reports that vendors may introduce *Residual* private APIs that are not invoked by any preloaded apps. In such cases, Bluebird is inherently limited. Besides, Bluebird is dependent on the accuracy of app-side sensitivity indicators extraction. We mitigate this threat by leveraging various methods, each tailored to identify a different indicator.

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

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

### 8 Related Works

**Android Access Control Analysis.** Framework access control has been extensively analyzed in the literature. Both static [2, 5, 6] and dynamic analysis approaches [8, 13] have been followed to infer API to access control mappings. Bluebird follows an approach similar to Axplorer [6] to extract framework-side access control and accordingly assign priors.

As mentioned in Section 5.6, our work is closely related to inconsistency analysis techniques, which identify access control anomalies in Android APIs through convergence analysis. Kratos [27], Acedroid [1], and ACMiner [16] detect inconsistencies by comparing access control enforcement among different paths leading to the same resource within the framework layer. AceDroid [1] improves the detection results of Kratos [27] (the earliest attempt) via access control normalization and modeling. ACMiner [16] is concerned with identifying security checks semi-automatically which Kratos and AceDroid define manually. IAceFinder [36] and FReD [18] detect a different category of access control inconsistencies: those that span different layers of the Android software stack; e.g., Native/Java layer inconsistencies [36], and Linux-file permissions/Java inconsistencies [18]. Poirot [11] re-conceptualizes inconsistency detection to account for uncertainty surrounding access control implementations in Android. It relies on various semantic and structural hints to connect resources to access control to resources. It hence addresses inaccuracies caused by over-approximations in [1, 16, 27].

Bluebird complements these existing works as discussed in Section 5.6. The gaps do not reflect a single access control inconsistency; rather a probabilistic consensus among (invoking) preloaded apps. As demonstrated in Section 5.6, Bluebird can exclusively unveil certain vulnerabilities that cannot be detected by other approaches.

**App Analysis.** A long line of research has been dedicated to Android app analyses. FlowDroid [4], EdgeMiner [7], Chex [23] precisely model the event-driven and asynchronous nature of Android apps. Bluebird benefits from the fundamental concepts of these tools to perform precise static analysis of preloaded apps.

**UI Analysis.** Prominent research works have been proposed to extract and analyze the UI of Android apps to infer security properties and flaws. Supor [17] identifies sensitive user input fields from an app’s UI using privacy-sensitive keywords, input field attributes and labels. UiPicker [25] does the same through a classification approach. [31] identifies sensitive widgets in an app’s UI by classifying used icons. These approaches have inspired our UI-driven analysis for prior probabilities of UI blocks.

**Probabilistic Inference in Security Applications.** Probabilistic techniques have been extensively used in security applications. LambdaNet [30] uses graph neural networks to probabilistically infer types for TypeScript. PRIMO [26] proposes an approach to utilize probabilistic inference imposed on static analysis to reduce the false positives and improve inter-component communication (ICC) resolution more efficiently. Hints among the source code such as variable names have also been used by several approaches to probabilistically infer program properties such as types [32] and physical unit consistency [19]. Lin et al. leverage probabilistic inference in reverse engineering [28]. Dietz et al. also leverage probabilistic inference to localize source code bugs [9]. Besides,
probabilistic techniques are widely used for binary analysis [24, 34], program enhancement [20], and vulnerability detection [33, 35].

9 Conclusion

In this paper, we propose Bluebird, a security auditing tool for Android that detects access control gaps in APIs, by learning security specifications from preloaded apps. Bluebird addresses various challenges hindering the process by combining program analysis, NLP techniques, and probabilistic inference. Our evaluation on 14 Android ROMs shows that Bluebird can uncover access control gaps. We were able to identify 11 new vulnerabilities out of the reported gaps.

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 RIPLE 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


[16] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden,


[35] Lin Zhiqiang, Rhee Junghwan, Wu Chao, Zhang Xiangyu, and Xu Dongyan. Discovering semantic data of


A Appendix

A.1 Additional Case Studies of Access Control Gaps

Manipulating wifi settings. Bluebird reported an access control gap in Samsung’s WifiService, setWifiSharingEnabled. As hinted by its name, the API allows changing custom Wifi Sharing settings. We manually investigated the case and found that if the API’s implementation enforces a normal-level permission android.permission.ACCESS_WIFI_STATE. We further checked its app-side constraints, generated by Bluebird in three preloaded apps (SecSettings, SystemUI, and SystemUIDesktop). The constraints all propagate a high sensitivity estimation from the app entries to the API, which we identified to be related to a protected broadcast interest from un-mappable memory with confidence. In Proceedings of the 19th Network and Distributed System Security Symposium, NDSS, volume 12, 2012.


A.1 Additional Case Studies of Access Control Gaps

Manipulating wifi settings. Bluebird reported an access control gap in Samsung’s WifiService, setWifiSharingEnabled. As hinted by its name, the API allows changing custom Wifi Sharing settings. We manually investigated the case and found that if the API’s implementation enforces a normal-level permission android.permission.ACCESS_WIFI_STATE. We further checked its app-side constraints, generated by Bluebird in three preloaded apps (SecSettings, SystemUI, and SystemUIDesktop). The constraints all propagate a high sensitivity estimation from the app entries to the API, which we identified to be related to a protected broadcast interest from un-mappable memory with confidence. In Proceedings of the 19th Network and Distributed System Security Symposium, NDSS, volume 12, 2012.


A.2 ProbLog Rules for Table 1

```
% 'Api' denotes an audited framework API.
% 'Ep' denotes an identified app entry discovered through app analysis.
% 'Id' denotes a unique identifier assigned to each observation.
% Framework-side Evidences:
% Rule 1:
% The random variable 'api_ac' indicates the probability of 'Api' being sensitive.
% 'fw_no_ac' denotes the observation that an API enforces weak or no access control.
% 0.1 :: api_ac(Api) :- fw_no_ac(Api, Id).
% 0.9 :: api_ac(Api) :- ep_ac(Ep), non_modifiable_context(Ep, Api, Id).
% 0.9 :: api_ac(Api) :- ep_ac(Ep), modifiable_context(Ep, Api, Id).
% Framework-side Evidences:
% Rule 2 and 3:
% The random variable 'ep_ac' indicates the probability of 'Ep' being sensitive.
% 0.9 :: ep_ac(Ep) :- ep_ac(Ep, Id).
% 0.9 :: ep_ac(Ep) :- app_ep_ac(Ep, Id).
% 0.8 :: ep_ac(Ep) :- ep_alert_dialog(Ep, Id).
% 0.9 :: ep_ac(Ep) :- ep_alert_dialog(Ep, Id).
% Rule 4:
% 'app_alert_dialog' denotes that observation that 'Ep' is a sensitive UI type (e.g., alert dialog)
% 'ep_alert_dialog' denotes that the probability of 'Ep' being sensitive
% because it is a sensitive UI type.
% 0.9 :: ep_alert_dialog(Ep) :- app_alert_dialog(Ep, Id).
% 0.8 :: ep_alert_dialog(Ep) :- ep_alert_dialog(Ep, Id).
% Rule 5, 6, and 7 (constructed similarly to Rule 4 above):
% 0.9 :: ep_non_cancellable_alert_dialog(Ep) :- app_non_cancellable_alert_dialog(Ep, Id).
% 0.8 :: ep_non_cancellable_alert_dialog(Ep) :- ep_non_cancellable_alert_dialog(Ep, Id).
% 0.9 :: ep_multi_confirm(Ep) :- ep_multi_confirm(Ep, Id).
% 0.9 :: ep_multi_confirm(Ep) :- app_multi_confirm(Ep, Id).
% 0.8 :: ep_multi_confirm(Ep) :- ep_multi_confirm(Ep, Id).
% 0.9 :: ep_multi_confirm(Ep) :- app_multi_confirm(Ep, Id).
% 0.8 :: ep_multi_confirm(Ep) :- ep_multi_confirm(Ep, Id).
% App-side Clues:
% Rule 8:
% 'nlp_contribution' denotes the observation that 'Api' is relevant to 'Ep'
% with a computed contribution score 'CT' (defined in Section 4.3)
% 'CT_p' denotes the (computed) implication probability of 'Api' being sensitive based on its contribution score to 'Ep'; Specifically, CT_p
% = CTScore * 0.9
% CT_p :: api_ac(Api) :- ep_ac(Ep), nlp_contribution(Ep, Api, Ct, Id).
% Rules 9 to 12 (constructed similarly to the implication constraint depicted in Rule 8):
% 0.9 :: \api_ac(Api) :- ep_no_ac(Ep), modifiable_context(Ep, Api, Id).
% 0.9 :: api_ac(Api) :- ep_ac(Ep), modifiable_context(Ep, Api, Id).
% Rule 10:
% 'non_modifiable_context' denotes the observation that the context of 'Api' is non-modifiable in 'Ep'
% 0.9 :: api_ac(Api) :- ep_no_ac(Ep), non_modifiable_context(Ep, Api, Id).
% 0.9 :: api_ac(Api) :- ep_ac(Ep), non_modifiable_context(Ep, Api, Id).
% Rule 11:
```
A.3 Samples of APIs from Synthetic Dataset

Here, we provide a sample of the APIs that implement strong access control and that were selected for the synthetic dataset construction, discussed in Section 5.3. The samples were chosen from Pixel 6 (V.11).

- FaceService.remove
- FaceService.setFeature
- ColorDisplayService.setSaturationLevel
- NetworkScoreService.requestScores
- SystemUpdateManagerService.retrieveSystemUpdateInfo
- StorageStatsService.queryStatsForUser
- StorageManagerService.decryptStorage
- StorageManagerService.encryptStorage
- MediaRouterService.getSystemSessionInfo