AppScopes: Fine-grained Access Control for App Collaboration on Android

Sanket Goutam
Stony Brook University

Kasra Kakaee
Stony Brook University

Amir Rahmati
Stony Brook University

***Abstract*—Android's application ecosystem increasingly relies on cross-app collaboration, yet the mechanisms that enable these interactions remain poorly understood and weakly controlled. Applications discover other apps through package visibility declarations and interact with them through embedded SDKs, such as social login libraries, enterprise management platforms, and analytics frameworks. While recent platform changes restrict broad app discovery, they do not address how applications exercise control once targets are known. In practice, SDK-mediated interactions operate under coarse, credential-based authority (*e.g.,* API keys or license tokens) that is opaque to the Android framework and not scoped to specific targets.**

In this paper, we present a systematic study of Android's app-collaboration layer, decomposing it into two components: *discovery* (which apps can be identified) and *control* (how apps can act on others). An analysis of $9{,}539$ popular Play Store applications reveals widespread overbroad discovery patterns, including apps querying large sets of unrelated targets. We further show that widely deployed SDKs adopt coarse-grained privilege models that lack per-target access control, enabling unintended or excessive cross-app interactions.

Motivated by these observations, we propose *AppScopes*, a fine-grained access control framework that introduces *collaboration scopes* as a first-class primitive for governing cross-app interactions. AppScopes requires applications to declare their functional collaboration domains and enforces access based on the intersection of these declarations, mediating both discovery and control. We implement a prototype using the Samsung Knox platform and evaluate it across three device generations. Our results show that AppScopes introduces minimal overhead; ≈30 µs per call on recent hardware and ≈0.2 ms on seven-year-old devices with a negligible memory footprint (28 bytes per wrapper) demonstrate the practicality of enforcing function-driven access control for cross-app collaboration.

***Keywords*—Android security, App collaboration, App intents, App permissions, Access Control.**

I. Introduction

Modern Android applications rarely operate in isolation. They routinely discover other installed apps, communicate through the Intent framework, and rely on embedded SDKs to enable cross-app functionality such as authentication, device management, and data sharing. While these mechanisms are essential for building rich application ecosystems, they collectively introduce a form of *ambient authority*: once an application satisfies coarse conditions (such as declaring a manifest entry or holding an SDK license), it can interact with other apps without further platform-level mediation. These interactions are largely invisible to users and weakly constrained by the Android security model [1].

Historically, Android imposed few restrictions on app discovery. Prior to Android 11, any application could enumerate all installed packages via `PackageManager` APIs without requiring explicit permission [2]. In response, Google introduced package visibility restrictions through the `<queries>` mechanism [3], requiring developers to declare intended discovery targets explicitly. While this change reduced unrestricted enumeration, it did not eliminate overbroad visibility. Applications can still declare large sets of specific package names, effectively reconstructing a detailed view of the user's installed software. In practice, this mechanism enables profiling behaviors that are disproportionate to an app's functionality, without triggering any sensitive permission checks [4].

Beyond discovery, SDK-mediated collaboration introduces a more opaque and less regulated control channel. Rather than relying solely on platform-mediated IPC, many SDKs embed privileged logic within host applications. As a result, authority is delegated through external credentials such as license keys, API tokens, or backend validation, rather than through Android's permission system. This design enables powerful cross-app capabilities but removes them from OS-level visibility and enforcement. Prior work has shown that such SDKs frequently operate with excessive privilege [5], and recent disclosures — ranging from enterprise SDK supply-chain attacks [6, 7] to intent-redirection vulnerabilities that expose cross-app flows at scale [8] — illustrate real-world consequences. In each case, the Android framework remains unaware of how authority is exercised across applications.

In this work, we present a systematic study of Android's app-collaboration layer as a unified attack surface, structured around two components: *discovery* (how applications identify one another) and *control* (how authority is exercised once connected). For the discovery layer, we statically analyze $9{,}539$ popular Play Store applications and recover $19{,}193$ queried package names, identifying 53 apps whose discovery behavior is inconsistent with their stated functionality. For the control layer, we analyze representative collaboration SDKs and find that they consistently adopt coarse-grained, credential-based access models that are not scoped or enforced by the Android platform.

Motivated by these findings, we propose *AppScopes*, a fine-grained access control framework that introduces *collaboration scopes* as an explicit abstraction for governing inter-app interactions. Applications declare their participation in functional domains via manifest metadata, and the system permits interaction only when participating apps share at least one common scope. This model enforces mutual consent at both the discovery and control stages, restoring platform-level visibility and enabling principled enforcement of cross-app communication.

We implement a prototype of AppScopes using the Samsung Knox platform, which provides a close approximation of framework-level control through its privileged APIs. Evaluating across three Samsung device generations, we show that AppScopes incurs minimal overhead: per-call latency increases by $\approx$30 µs on recent hardware and $\approx$0.2 ms on seven-year-old devices, with a negligible memory footprint.

Contributions. This paper makes the following contributions:

- We conduct a large-scale empirical study of Android's discovery layer, revealing widespread overbroad query behavior across popular applications (§III).
- We characterize the access control models of widely deployed collaboration SDKs, showing that they rely on opaque, credential-based mechanisms that bypass OS-level enforcement (§III).
- We design, implement, and evaluate AppScopes, a scope-based mediation framework that enforces fine-grained, auditable, and function-driven control over cross-app interactions (§V, §VII).

II. Background

Android applications execute in a strongly sandboxed environment: each installed package runs in its own process and is assigned a distinct Linux user identifier (UID) at install time, and sensitive resources are exposed only through user-granted permissions and framework APIs. Process isolation, UID-based access control, and platform hardening (*e.g.,* SELinux and the Android Runtime) together form the baseline app sandbox that underpins Android's security model [1, 9, 10].

To enable collaboration between sandboxed apps, Android exposes OS-mediated IPC primitives (such as the Intent framework, Binder RPCs, Content Providers, and package-manager APIs) that let apps discover other installed apps, invoke functionality, and share data. These primitives are visible to and mediated by the platform, meaning that the OS resolves app intents, enforces exported-component and provider permissions, and controls which packages an app can query or enumerate.

Over the last decade SDK-mediated collaboration has grown alongside platform APIs. Rather than routing interactions through platform IPC channels alone, many apps started embedding first-party and third-party SDKs directly in their process. The SDK runs with its host app's privileges, implements its own discovery and authorization semantics (API keys, OAuth tokens, licensing), and can enable cross-application functionality that the OS cannot directly observe or mediate.

OS-mediated discovery and IPC. The platform primitives provide two essential capabilities. First, discovery: the ability for an app to learn about other installed apps (such as package visibility, intent resolution). Second, controlled interaction: the ability to invoke another app's component (explicit and implicit intents), bind to long-lived services (Binder/AIDL), or access provider-backed structured data (Content Providers). Because these channels are mediated by the OS, permission checks, visibility filters, and exported-component constraints can limit and audit cross-app flows.

Recent platform changes have narrowed some of the discovery surface. For example, Android 11 (API 30) introduced package-visibility filtering, forcing apps to declare explicit `<queries>` or to request the restricted `QUERY_ALL_PACKAGES` permission to enumerate other apps [3]. These changes reduce unbounded enumeration but do not change the fact that OS-mediated IPC remains the visible, auditable channel that the original permission model was built around.

SDK-mediated collaboration. The use of app-centric SDKs shifts discovery and control into application space. A broad and representative set of collaboration SDKs is available to developers today, including:

- Google Play services and platform SDKs that offer identity, location, and cross-app APIs.
- Enterprise mobility and MDM SDKs (*e.g.,* Samsung Knox) that can configure devices and manage applications via privileged APIs.
- Social and single-sign-on SDKs that broker authentication and token exchange across integrating apps.
- Health and sensor-data SDKs (*e.g.,* Health Connect) that aggregate physiological signals from users' devices (including wearables) and expose derived metrics to multiple clients.
- Advertising and analytics SDKs that collect telemetry and enable cross-app profiling and targeting.

Because these SDKs execute inside host apps, they inherit the app's permissions and can implement cross-app sharing semantics that remain opaque to the OS. An SDK-mediated exchange between two apps therefore appears to the platform as two apps independently interacting with their embedded SDK code rather than as an auditable channel of data flow between the apps themselves. While this architecture enables flexible integration, it also creates the potential for undisclosed user-data collection and sharing.

Android recognised this emerging mismatch and proposed the Privacy Sandbox SDK Runtime, a dedicated runtime intended to sandbox third-party SDKs and grant them explicitly scoped data-access and permission semantics in order to restore platform-level safeguards [11]. Despite plans to introduce the SDK Runtime with Android 13, the proposal was deprecated in October 2025 after developer pushback about increased friction and limited privacy benefits [11, 12]. Without a platform-enforced SDK runtime, many SDK-mediated flows remain outside the OS's direct control.

Problem Statement. The migration from an OS-centric IPC model to an SDK-centric collaboration model undermines a core assumption of Android's permission architecture: that cross-app communication occurs over platform-mediated channels which the OS can observe and constrain. With the SDK Runtime no longer on the platform roadmap, privileges and authorization semantics have fully migrated to the application layer, where the platform cannot reliably enumerate participants, scope authority at a per-target level, or enforce policy decisions.

To reason about these risks we propose a simple taxonomy: *app discovery* mechanisms (how an app finds other software on the device) and *app control* mechanisms (which confer operational authority, such as device-management APIs, identity

tokens, or health-data scopes). SDKs can participate in both layers, often broadening discovery and centralising control in ways that evade OS visibility. We substantiate these claims through a case study in §III.

III. Case Study: App Collaboration in the Wild

To illustrate the privacy implications of both layers identified in §II, we conduct a case study of cross-app collaboration in the wild. We study the *discovery* layer quantitatively through a large-scale analysis of manifest-based package-visibility declarations across 9,539 popular Play Store apps, characterising cross-app discovery behaviour by category and identifying apps querying well beyond the scope of their stated functionality. Because SDK-mediated mechanisms operate inside application code and resist tractable static analysis, we study the *control* layer qualitatively by examining the access-control models of representative collaboration SDKs.

Prior work by Gagel et al. shed some light on this issue while restricting themselves to only the top 2,000 Google Play apps [13]. They found 21.6% of apps in their dataset to exploit package visibility loophole to access user's installed apps. We extend their analysis to more broadly cover and categorize the behavior seen across apps from different regions and categories.

A. Dataset and Methodology

We assembled a corpus of 9,539 APKs by combining top-ranked apps in each Play Store category, top apps from ten large geographic regions (US, India, Japan, Indonesia, Brazil, Canada, Australia, UK, Mexico, China), and keyword-driven search results. We limited the study to free apps with publicly available APKs indexed by AndroZoo [14], spanning a collection period over 5 months until July 2025. Our entire pipeline for data collection and analysis is shown in Figure 1.

Each APK was statically analyzed with *apktool* to decode the AndroidManifest.xml and extract any `<queries>` declarations (Figure 1). We focus on manifest-encoded discovery because it is explicit and tractable via static analysis; SDK-mediated discovery typically occurs inside application code and is substantially harder to recover.

Our analysis identifies two sets: querying apps (α), which declare one or more package names in `<queries>`, and the union of referenced package names (β). We write $\alpha \rightarrow \beta$ for the querying relation ("α queries β"). From the corpus of 9,539 apps we find 4,786 querying apps in α and 19,193 distinct package names in β; 185 apps in α request the restricted `QUERY_ALL_PACKAGES` permission.

To characterize category-level behavior we map Play Store categories into 13 broad groups (for example, "Fitness" and "Medical" are grouped as "Health"). Classifying package names in β is challenging because many are not present in Play Store. We therefore apply a three-stage classification: (1) Play Store lookup, (2) keyword-based filtering on package names (for example, package names containing substrings like "media"), and (3) LLM-assisted classification (using the Llama 3.1 8B model) with web search (via the DuckDuckGo API) for remaining unknowns. Table II summarizes dataset statistics and classification coverage.

B. Characterizing Package Discovery

Figure 3 shows how querying apps (α) and queried packages (β) are distributed across the 13 categories. Two patterns are noticeable in the left half of the figure. First, most query edges ($\alpha \rightarrow \beta$) terminate in a small set of popular categories such as social and travel apps. Second, a few categories (notably finance) are queried heavily but from only a handful of α-side apps. The first pattern reflects targeted discovery of well-known packages for a genuine use case; the second suggests broad reconnaissance of niche or regional packages — for instance, a gaming app profiling users based on the banking apps they use.

Figure 2 shows a heavy-tailed distribution of manifest queries: 309 apps declare more than 30 packages, roughly 150 apps declare more than 50, and a single app lists over 7,000 package names. In total, 16% of apps in the corpus declare more than ten packages. These observations indicate that manifest-based discovery is both widespread and capable of producing high-fidelity signals about which software is present on a device.

To identify potentially anomalous behavior we implement three complementary heuristics (H1–H3), apply them to the set of querying apps, and manually review every flagged app by examining its queried package list, Play Store description, and declared permissions to determine whether the queries have a plausible functional justification. Using this procedure we identify 53 applications with suspicious or unexplained query lists; Table I summarizes representative patterns. The full list of flagged apps and per-heuristic triggers is released alongside this paper to support reproducibility.

- H1 **Category-level deviation.** For each category of apps in α we compute the baseline distribution of queried categories in β — *i.e.,* for an app in α of category X, the likelihood of querying a β-side app of category Y. Any app whose per-category query distribution visibly deviates from this baseline is forwarded to manual review, since such deviations suggest discovery behavior outside the norm for its category.
- H2 **Cross-risk querying.** We classify categories along a risk spectrum based on the sensitivity of the data they typically handle: finance, banking, health, and identity/wallet apps are treated as high-risk, whereas gaming, news, and messaging apps are treated as low-risk. We flag any querying app whose β-side targets contain more than 10% high-risk apps.
- H3 **Volume-based outliers.** We flag any app that declares more than 30 packages in its `<queries>` block, the point at which the query distribution (Figure 2) visibly transitions into the tail.

C. Characterizing SDK-Mediated Collaboration

Beyond the discovery layer, SDK-mediated collaboration grants integrating apps operational authority over other apps and user data through access-control models that are entirely opaque to the Android framework. Rather than routing cross-app authority through platform APIs (*e.g.,* `PackageManager`, declared intents), SDKs embed proprietary privilege mechanisms and backend-assisted coordination. Each host app embeds its own copy of the SDK and executes

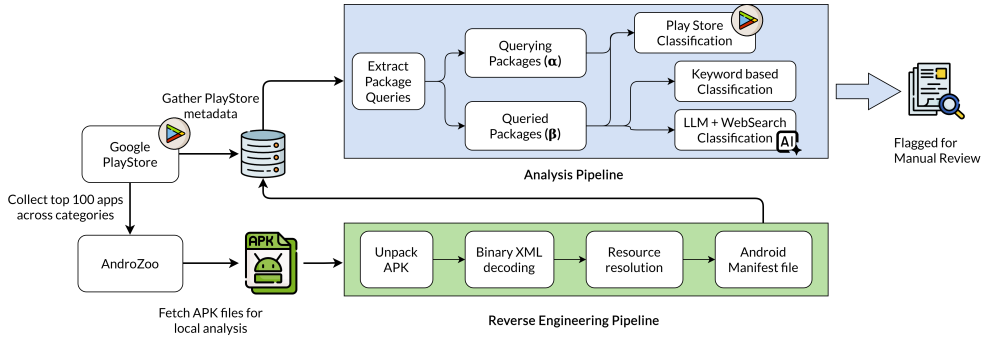

Fig. 1. Our app analysis pipeline for investigating the package-manager-based app discovery mechanism used in the wild. We identify sets of popular Google Play Store apps (α) that attempt to discover the presence of a set of queried apps (β) on the user's device, and characterize the querying relation $\alpha \rightarrow \beta$.

TABLE I. REPRESENTATIVE APPS WITH SUSPICIOUS QUERY PATTERNS FLAGGED BY H1–H3. FOR EACH TARGETED DOMAIN, WE NOTE A LEGITIMATE RATIONALE AND A CONCRETE ANOMALOUS EXAMPLE FROM OUR CORPUS.

Targeted Domain	**App Count**	**Reasoning and Example**
Crypto/Stock Trading	25	*Legitimate:* Portfolio aggregators or tax apps may need to discover trading apps to import data. *Anomalous:* MiChat (Messaging App) queries 5 crypto/stock apps. Messaging apps have no clear reason to map a user's financial portfolio.
Finance/Banking	23	*Legitimate:* Budgeting or e-commerce apps may check for digital wallets or bank apps to enable payments. *Anomalous:* eWeather HDF (Weather App) queries 153 banking/micro-loan apps. Weather apps do not require knowledge of a user's financial apps, suggesting profiling.
Root/Emulator Detection	14	*Legitimate:* Security-sensitive apps (banking, DRM, MDM) may check for root/emulator status to protect data. *Anomalous:* Breaking News (News App) queries 8 root/superuser apps. News readers have no valid reason to check device integrity.
Debug/Beta SDKs	16	*Legitimate:* Developer tools or bug bounty platforms may look for debug/test builds for integration or testing. *Anomalous:* Ullu (Video Streaming App) queries debug versions of popular apps (*e.g.,* Truecaller), likely to fingerprint developer devices or probe for exploits.

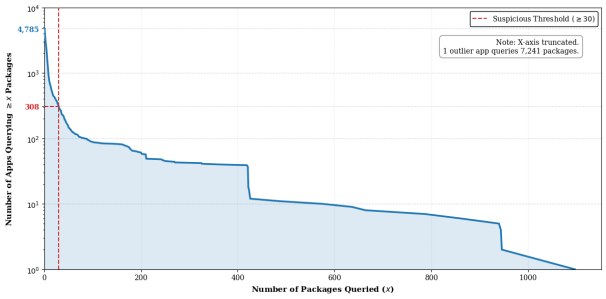

Fig. 2. Heavy-tailed distribution of packages declared per app in `<queries>`. 309 apps in α query more than 30 packages; a single outlier lists over 7,000.

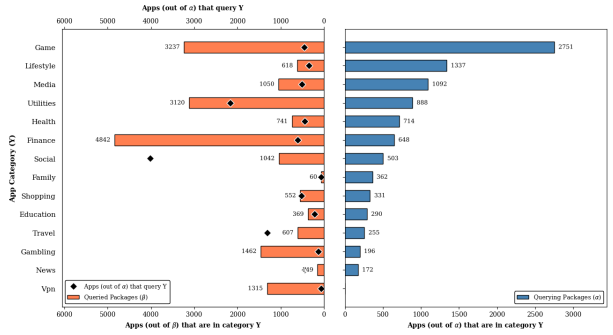

Fig. 3. Category distribution of querying apps (α, right) and queried package names (β, left).

it under its own UID, so cross-app "collaboration" is emulated through backend aggregation and shared identifiers rather than direct OS-mediated interaction — effectively shifting the trust boundary away from the device and into remote infrastructure.

Enterprise mobility SDKs (*e.g., Samsung Knox* [15]) exemplify this pattern in its most extreme form: an OEM-issued license authorises enumeration and control of every installed app via privileged APIs, with users granting only coarse device-administration consent. Identity and social SDKs (*Facebook SDK* [16], *Sign in with Google* [17]) mediate authority through OAuth tokens and intent probing; advertising and analytics SDKs (*Google Mobile Ads* [18], *Firebase Analytics* [19]) reconstruct a logical cross-app layer through shared backend services; and even privacy-oriented frameworks such as *Health Connect* [20] mediate data access through SDK interfaces rather than explicit IPC channels. Across categories, cross-app authority flows through SDK-internal credentials and shared backend infrastructure that the OS cannot enumerate or constrain. Prior work has explored SDK isolation and runtime monitoring to close this gap [5, 11], but adoption has been limited [12].

IV. THREAT MODEL

As established in §II, Android's permission model assumes cross-app interactions occur over OS-mediated channels, where the platform can apply permission checks and audit trails. SDK-mediated collaboration violates this assumption: an SDK executing inside a host app inherits its UID and Binder credentials, while cross-app authority flows through application-layer credentials (license keys, OAuth tokens, API

keys) that the OS cannot enumerate or constrain. The threat model below characterizes who can exploit this gap and how.

Adversary model. We model a *compliant but opportunistic adversary*: an app or embedded SDK distributed through the Google Play Store that abuses the structural gap above to exceed the authority implied by its stated functionality. This adversary differs from a traditional malware author in three respects: it passes the app-store review, operates inside a legitimately installed process, and exploits *permitted* SDK APIs rather than OS vulnerabilities.

We exclude adversaries who (i) compromise the Android kernel or firmware, (ii) tamper with the app-installation pipeline, or (iii) subvert the SDK vendor's own backend infrastructure. Such attacks require OS-level or supply-chain defenses beyond the scope of an application-layer access control framework.

We identify two threat surfaces corresponding to the discovery and control layers of §II.

T1 ***Discovery-layer reconnaissance and user profiling.*** The `<queries>` manifest element permits enumeration of installed packages without platform-imposed limits. Aggregating such visibility yields high-fidelity fingerprints (for example, presence of finance, health, or religious apps) without requesting sensitive permissions. In our study (§III) we observed a weather app querying 153 banking apps, a messaging app querying five cryptocurrency apps, and a single app listing over 7,000 package names. With `QUERY_ALL_PACKAGES`, an adversary can refresh and correlate these fingerprints against backend profiles silently, leaving no OS audit trail.

T2 ***Control-layer exploitation of overprivileged SDK access.***
Device-management abuse. Enterprise mobility SDKs (*e.g.,* Samsung Knox) grant licensed hosts powerful device-management capabilities (force-stop, wipe, uninstall) via OEM-issued license keys; the Android framework does not represent the intended management scope, creating tangible organizational risks as evidenced in prior supply-chain attacks [6].
Cross-app data harvesting. Once such cross-app authority is held, the same credentials can be turned against other installed apps to enumerate, inspect, or exfiltrate their data. CISA's *Stryker* advisory documents this at scale: adversaries abused Microsoft Intune's management APIs to inventory and exfiltrate data from applications across managed endpoints, leaving no OS-level audit trail [7]. Because the authority flowed through SDK credentials rather than platform permissions, Android had no basis on which to mediate or audit the access.

Scope and Assumptions. AppScopes addresses T1 and T2 as an *additional* authorization layer on top of Android's existing permission model. Within T2, we target overreach that crosses UID boundaries: a host application (or an SDK running under its UID) exercising authority over *other* installed apps. Same-UID concerns such as a library harvesting data from its own host (*e.g.,* OneAudience [21]) are the complementary target of process-level SDK isolation [5, 22] and fall outside our scope. AppScopes likewise does not protect apps that decline to declare scopes: a legacy or hostile app that does not opt in remains outside the control plane, though Android's existing sandboxing continues to prevent it from affecting other installed apps.

Two assumptions underpin enforcement. First, **honest manifest declarations**: participating apps declare their collaboration scopes faithfully; a non-opting app is outside the control plane by construction, and closing that gap requires app-store or OS-level policy enforcement. Second, **trusted installer**: manifest metadata is trusted as delivered by the installer; a production deployment requiring stronger guarantees would sign scope declarations under a platform key (§V-B).

V. AppScopes

We propose *AppScopes*, a permission control mechanism that enables declarative and policy-driven inter-app collaboration based on functional grouping. In contrast to existing Android mechanisms, where inter-app interactions are mediated either through coarse-grained permissions or implicit IPC channels, AppScopes introduces an explicit declaration model in which applications specify a *collaboration ID* representing a well-defined functional domain. Applications that implement related functionality may opt into the same identifier, thereby signaling their intent to participate in a shared interaction context.

This model is analogous to group-based access control primitives such as Unix supplementary groups: scopes act as capabilities that a caller and a target must mutually hold for any interaction to proceed, rather than privileges that either party can unilaterally grant itself. By mapping applications into logically isolated functional domains, AppScopes enables the operating system to enforce fine-grained interaction policies based on functional necessity rather than broad privileges. Participation in a given scope is explicitly declared and mediated by the platform, ensuring that inter-application communication occurs only among mutually consenting members of the same functional group.

A. Design Goals

Our measurement study in §III highlights three fundamental limitations of app collaboration. We restate them here as findings (F1–F3) and map each to a corresponding design goal (G1–G3) that informs AppScopes.

- ***F1 ⇒ G1: Deterministic app interactions.*** Our study found that apps routinely attempt to discover peers well beyond their stated functionality, with no limit on the breadth of a `<queries>` declaration (§III). AppScopes must ensure that discovering the presence of other installed apps must be deterministic and bounded.
- ***F2 ⇒ G2: Platform-mediated access.*** Today, cross-app authority flows through credential-based mechanisms (license keys, OAuth tokens, API keys) that the OS cannot observe or constrain; the "intent redirection vulnerability" recently disclosed by Microsoft Research exemplifies this gap, enabling SDK-mediated flows between apps without platform mediation [8].

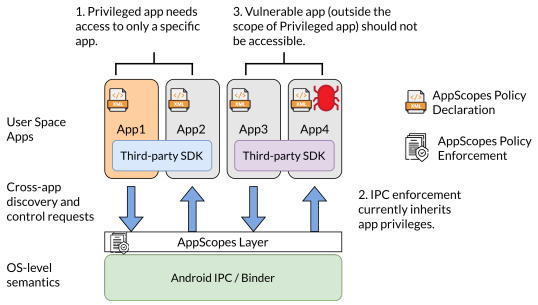

Fig. 4. AppScopes is designed to exist as a wrapper on top of Android framework IPCs and Binder functionality. Apps (along with third-party SDKs) primarily rely on framework layer bindings to perform any action outside of their sandbox. AppScopes acts as the moderator for all calls to the framework layer APIs.

AppScopes must re-anchor every cross-app authorization decision in a check the platform owns, so that enforcement and auditing do not depend on SDK-internal logic. Exposing this mediation surface to end users via a revocation UI is compatible with, but orthogonal to, this work.

- ***F3 ⇒ G3: Fine-grained and auditable policy.*** All collaboration SDKs today authorize through opaque mechanisms (license keys, API tokens, signed app identities) whose decisions are masked to the OS and third-party auditors. We posit that a declarative policy specification is necessary that allows auditors and app stores to statically extract from an APK and verify against the app's observed behavior.

B. Architecture Overview

Recall from §II that Android attributes every cross-app interaction to the calling application's UID, regardless of whether the action originates in first-party code or an embedded SDK [23]. Enforcement is therefore limited to the permissions granted to that UID, with no visibility into the functional context driving the interaction. This abstraction gap is what AppScopes closes: the framework validates requests by caller identity, but has no basis on which to ask whether the *purpose* of the request is consistent with either party's declared role.

To address this limitation, we introduce *functional intent* as a first-class objective with AppScopes. Specifically, we identify that there exists a disconnect between the functional requirements that motivate inter-app interactions and the coarse-grained permission model used to authorize them. Our system, AppScopes, bridges this gap by introducing an explicit declaration of collaboration intent at the application level using app's manifest files. Under this model, applications must declare the functional domains in which they intend to participate (*e.g.,* gaming, finance, health), and the system enforces interaction policies based on these declarations.

In its envisioned platform-integrated form, AppScopes introduces a mediation layer that intercepts and regulates cross-app IPC, organizing applications into logically isolated groups defined by shared functional identifiers. Our prototype (§VI) realizes this layer at the SDK-wrapper level for Knox; a full AOSP integration is deferred to future work. Applications within the same group are permitted to interact using standard IPC mechanisms, while interactions across groups are denied by default unless explicitly authorized. For example, a gaming

```
// Manifest declaration (simplified)
<meta-data
  android:name="com.example.appscopes.<
       COLLABORATION_ID>"
  android:value="group-1,group-2" />

// Policy enforcement (simplified)
Set<String> callerScopes = getScopes(callerPkg);
Set<String> targetScopes = getScopes(targetPkg);

Set<String> intersection = new HashSet<>(
     callerScopes);
intersection.retainAll(targetScopes);

if (intersection.isEmpty()) {
   throw new SecurityException("AppScopes: access
        denied");
}

// Forward to underlying IPC
invokePrivilegedOperation(targetPkg);
```

Fig. 5. AppScopes policy model. *<COLLABORATION_ID>* is a developer-declared identifier representing a functional group. Access is granted only if the caller and target share at least one collaboration ID, computed via set intersection.

application that requires access to financial services for in-app purchases must declare this functionality in its manifest, making the interaction visible to the framework. The system can then permit controlled communication between the gaming and finance domains. In the absence of such a declaration, cross-domain interactions are blocked, enforcing a default-deny policy.

The AppScopes architecture (Figure 4) is centered around a single abstraction, the *collaboration ID-based scoping*, and two enforcement mechanisms that regulate cross-application interaction at both discovery and control stages.

A collaboration ID is a developer-declared, set-valued identifier attached to an application via Android manifest. Unlike traditional single-valued labels, scopes are inherently compositional: an application may belong to multiple functional domains simultaneously (*e.g.,* enterprise management and health data sharing) based on its functional requirements. The core authorization primitive then is to simply compute a set intersection over the defined scopes. Given a caller c and a target t, interaction is permitted only if their declared scope sets overlap, *i.e.,* $scope(c) \cap scope(t) \neq \varnothing$. This ensures that apps collaborate over mutual consent – neither the caller nor the target can unilaterally establish a communication channel without explicitly sharing membership in at least one collaboration group.

Permission enforcement (G1,G2). AppScopes enforces this model through two independent enforcement layers. The first layer regulates *app discovery*. Rather than allowing unrestricted visibility via `QUERY_ALL_PACKAGES`, applications declare a specific intent action (*e.g.,* `ACTION_COLLABORATOR`) within a `<queries>` block. Only applications that explicitly expose a matching `intent-filter` are discoverable, ensuring that visibility is restricted to opt-in participants. As scope metadata is co-declared with the intent filter, the set of discoverable applications is inherently bounded by those that have explicitly joined the collaboration group.

The second layer governs *app control*. All privileged SDK interactions are routed through a lightweight policy wrapper that mediates access before invoking the underlying framework API. At runtime, the wrapper retrieves the collaboration

scopes of both the caller and the target via the PackageManager, computes their intersection, and denies the request if no common scope exists. These two layers operate orthogonally *i.e.,* even if a caller learns about a target through an external channel, it cannot exercise control unless the scope condition is satisfied. Conversely, undiscoverable apps remain inaccessible regardless of scope alignment.

Integration with Android. AppScopes is designed to integrate directly with Android's existing security mechanisms rather than replace them. Platform-level enforcement, including signature permissions, SELinux isolation, and package visibility filtering, remains intact. The scope check acts as an additional constraint layered above existing app authorization mechanisms, such as license validation, API key verification, or app certificate validation. Because AppScopes mediates interactions *between* app UIDs, SDKs embedded within a host application inherit that host's scope set; harvesting that occurs inside a single UID (*e.g.,* an analytics SDK exfiltrating host data) falls outside this layer and is the complementary target of process-level SDK isolation [5].

Auditability (G3). Finally, the declarative nature of collaboration scopes enables auditability. Because scopes are explicitly defined in application manifests, the effective collaboration graph can be reconstructed through static analysis without inspecting runtime behavior or reverse engineering embedded SDKs. This improves transparency for app store reviewers, enterprise administrators, and end users, enabling systematic reasoning about cross-application interactions.

VI. Implementation

We implement AppScopes as a Java library, `AppScopesPolicy`, positioned between a privileged (superuser) application and the Samsung Knox Application Policy interface. Our prototype is designed to approximate a framework-level deployment as closely as possible without modifying the Android Open Source Project (AOSP). OEM-provided enterprise SDKs such as *Samsung Knox SDK* [15] expose privileged device management APIs that operate with elevated authority, making them a suitable substrate for prototyping enforcement mechanisms that would otherwise require framework integration.

Design rationale. Knox provides a practical approximation of framework-level control: applications provisioned with a valid license can exercise system-wide authority over other apps via Binder-backed APIs. By interposing on these APIs, our implementation emulates how AppScopes would operate if integrated directly within the Android framework. This allows us to evaluate enforcement logic, performance overhead, and compatibility on commodity devices without requiring OS modifications.

Scope declarations and integration. Participating applications declare their collaboration scopes as manifest metadata and opt into discovery via a dedicated intent filter as shown in Figure 5. The superuser application additionally declares a constrained `<queries>` block to limit visibility to scope-aware participants. This mirrors how AppScopes would integrate with existing Android package visibility and IPC mechanisms.

Limitations. As a library-level prototype, our implementation inherits limitations of the underlying SDK model. In particular, scope declarations are not cryptographically bound to the platform, and enforcement can be bypassed by applications that directly invoke Knox APIs outside our wrapper. These limitations stem from the lack of control over the framework and Binder entry points. A production deployment would require integrating AppScopes into core system services (*e.g.,* `PackageManagerService`) or into OEM-managed Binder stubs to ensure mandatory enforcement.

TABLE II. Dataset snapshot for the package-discovery analysis.

Description	Count
Total APKs analysed	9,539
Apps declaring `<queries>` (α)	4,786
Apps in α that request `QUERY_ALL_PACKAGES`	185
Unique package names referenced (β)	19,193
Packages in β not found on Play Store	11,678
Packages classified by keyword filtering	8,140
Packages classified using LLM+web search	3,509
Remaining unclassified packages	29

Despite this, we select Knox as our implementation target because it represents an extreme case of SDK-mediated authority, where a single credential enables unrestricted cross-application control. This makes it an ideal testbed for evaluating AppScopes under maximal privilege conditions. However, the underlying design generalizes beyond enterprise SDKs. The same abstraction of scope-based mediation can be applied to other SDK categories, including social, analytics, and health frameworks, by interposing on their respective IPC or data access pathways. Ultimately, a complete realization of our system would reside within the Android framework itself, eliminating the need for per-SDK wrappers and ensuring uniform enforcement across all forms of cross-application interaction.

VII. Evaluation

We designed our evaluation to assess the practicality of our framework for real-world deployment. We first evaluate how adopting AppScopes affects the performance of an application in the form of additional latency for the authorization checks. We also evaluate the memory footprint needed for the additional package information validation.

Setup. Because our prototype is built on the Samsung Knox SDK, we are restricted to Knox-compatible hardware; we use a Samsung Galaxy Tab S6 (2019), Samsung Galaxy S25 (2025), and Samsung Galaxy S26+ (2026), spanning three platform generations across seven years of Samsung silicon. The AppScopes wrapper itself performs a constant-time set intersection over small scope sets, so we expect the overhead characterisation to generalise beyond Samsung devices; confirming this, however, requires a vendor-neutral framework-level deployment that we leave to future work.

Demo apps. On each device we install a Knox-licensed superuser app [15] alongside six dummy applications representing common categories (entertainment, banking, social, and so on). The Knox license grants the superuser app the ability to dump application info, force-stop, or uninstall any other app on the device. We assign each dummy app into one of four collaboration scopes (`group-1` through `group-4`) and enroll the superuser app into a subset of these groups, mimicking a narrowly scoped access-control model over the representative set of apps.

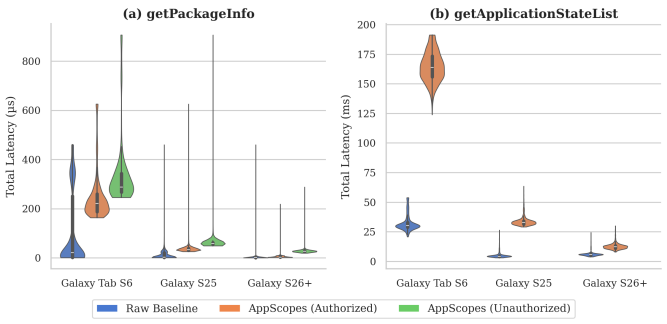

Fig. 6. Violin plots of total API call latency as observed on the three test devices. Each plot shows p1-p99 latency distribution.

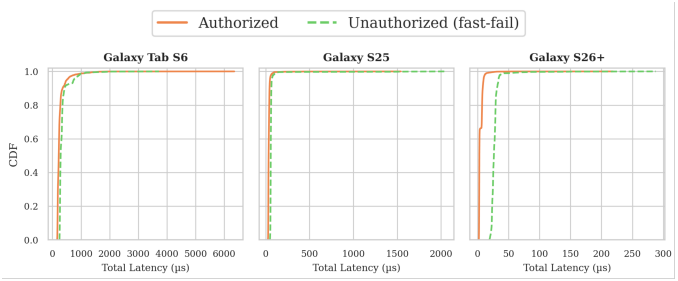

Fig. 7. Empirical CDF of total latency observed for `getPackageInfo` across the three devices for $N = 1000$ calls.

Benchmarking AppScopes. To benchmark our prototype, we develop an evaluation harness that invokes a target API $N = 1{,}000$ times and records the per-invocation latency. To isolate framework cost from JIT warm-up, we clip the top 1% of each distribution before reporting medians. For the baseline measurement we run the same API invocations directly against the Knox binder, bypassing the AppScopes layer.

We benchmark two APIs that are representative of the *app discovery* and *app control* taxonomy identified earlier. `getApplicationStateList` is a Knox API that returns the list of other installed apps on the device (*discovery*). `getPackageInfo` is provided by the PackageManagerService and returns detailed information about a requested package (*control*).

Figure 6 compares the observed latency for *app discovery* and *app control* under direct invocation (baseline) and through our AppScopes wrapper. For authorized callers, AppScopes adds ≈30 µs per call on the S25 and S26+, and ≈200 µs (0.2 ms) on the older Tab S6, across both APIs — a constant cost dominated by a single `PackageManager` metadata lookup and a two-set intersection over small scope sets. For unauthorized callers the latency is marginally higher due to audit logging for the rejected request; for `getApplicationStateList`, AppScopes short-circuits to an empty list. Figure 7 plots the empirical CDF of authorized versus unauthorized calls across the three devices, confirming that the two distributions remain separated by only tens of microseconds.

Memory footprint. We also measure the steady-state memory cost of an `AppScopesPolicy` instance by allocating 10,000 wrappers after a forced `Runtime.gc()` and dividing the resulting heap delta by the allocation count.

Across all three devices an `AppScopesPolicy` wrapper costs approximately 28 bytes of managed heap and zero native memory — a footprint lighter than a short Java `String`. A device-management process holding 1,000 concurrent wrappers (one per managed app) pays under 30 KB of total overhead, which we characterise as negligible for any production deployment.

Security enforcement. Beyond measuring cost, we verify that AppScopes enforces the properties motivated by our threat model. When the superuser app shares no scope with a target, calls to `getApplicationStateList` omit the target from the returned list, rendering it invisible at the discovery layer (mitigating T1); privileged operations such as `getPackageInfo` and `uninstallPackage` raise `SecurityException` before reaching the Knox binder (mitigating T2). For correctly scoped pairs the wrapper is transparent and preserves baseline semantics. We confirmed this across all six dummy apps on each device.

Package visibility minimisation. Beyond the policy-layer check, AppScopes tightens the discovery layer by replacing the superuser's `QUERY_ALL_PACKAGES` declaration with a targeted `<queries>` block scoped to the `ACTION_COLLABORATOR` intent. On each of our devices this change reduces the number of packages enumerable by the superuser from the full system package list (> 300 packages on a typical device) to exactly those apps that declare the reciprocal intent filter — six dummy apps in our deployment. Other user apps are not visible to the superuser app.

VIII. Discussion

Framework-level deployment. Our prototype interposes on Knox APIs at the library level. An app invoking Knox APIs directly can bypass AppScopes's enforcement. Mandatory enforcement would require integrating scope checks into `PackageManagerService` and the relevant Binder stubs — an engineering and OEM-adoption challenge rather than a conceptual one. We leave a full AOSP-integrated prototype, together with a Settings-level revocation UI that exposes the scope graph to end users, as future work.

Static analysis does not resolve intent. Our case study is grounded in manifest-declared `<queries>` entries, which are explicit and tractable statically. However, a query is not itself evidence of malicious intent — it may reflect a legitimate interoperability need, a bundled third-party SDK, or opportunistic reconnaissance. The 53 apps we flag are implausible given their stated functionality, not confirmed as policy violations. Resolving intent would require dynamic instrumentation or source-code analysis, neither of which is possible at scale. We consider it outside the scope of this work.

Off-device data aggregation. AppScopes governs on-device inter-app interactions. It does not prevent an app from transmitting legitimately collected data to a cloud backend, where cross-app profiles may be constructed at scale. That said, by restricting which on-device apps a privileged app can observe, AppScopes reduces the fidelity of the device-side signal before it leaves the device. Addressing off-device aggregation fully requires complementary mechanisms such as differential privacy at the SDK boundary, data-flow labelling, or regulatory

constraints, which are all orthogonal to and compatible with our scope-based model.

Deployment incentives. A scope-based model is compatible with incremental rollout. Non-opting apps retain current behaviour, so adoption does not require immediate migration. Platform operators can reuse the existing `PackageManagerService` permission-check path and manifest tooling. Application developers gain reduced attack surface from other apps at negligible runtime cost (§VII). This incremental path mirrors how the `<queries>` mechanism itself was deployed in Android 11.

IX. Related Work

Third-party library isolation. Prior work has extensively studied the security risks of third-party SDKs [24, 25, 26, 27, 28], particularly the problem of cross-library data harvesting (XLDH), where embedded libraries access and aggregate sensitive data across application boundaries [5]. Because SDKs execute within the host application's process and inherit its permissions, they effectively bypass privilege separation, enabling unrestricted access to sensitive resources. To mitigate this, systems such as AdSplit and FlexDroid propose isolating advertising and third-party libraries into separate processes with restricted permissions [26, 27].

App compartmentalization. Beyond library-level isolation, research has explored application-level compartmentalization to enforce least privilege within Android apps [29, 30, 31, 32]. Systems such as Boxify and MOSES introduce containerization and multi-persona environments to isolate application execution contexts and restrict inter-app data flows [30, 31]. TaintDroid-inspired extensions [33] and SEAndroid-based policies [34] further enforce isolation boundaries across application components. These approaches reduce the attack surface introduced by overprivileged applications, but typically operate at coarse granularity (*e.g.,* app or container level) and do not directly address fine-grained, function-driven inter-app collaboration.

Taint analysis and information-flow tracking. Dynamic and static taint analysis techniques have been widely adopted to detect privacy leaks in Android applications [35, 36, 37]. TaintDroid provides real-time tracking of sensitive data flows across app components and system interfaces [33], while FlowDroid and Amandroid extend this model with static analysis to identify potential data leaks at scale [38, 39]. These systems offer strong visibility into how data propagates within and across applications, but they are primarily detection mechanisms and do not enforce access-control policies, leaving a gap between identifying and preventing cross-app data misuse.

Discovery-layer measurement and installed-app fingerprinting. Closest in spirit to our §III measurement is a line of work showing that the set of applications installed on a device is itself a high-fidelity identifier: Achara *et al.* demonstrate that small subsets of installed apps uniquely identify individual users [40], and Kurtz *et al.* extend this to device-level fingerprinting from personalised configurations [41]. Practitioner reports have further documented the persistence of overbroad `<queries>` declarations after Android 11 [4]. Our work complements this literature by providing a large-scale academic measurement of the post-Android-11 `<queries>` discovery surface, connecting observed query patterns directly to the T1 profiling threat.

Inter-app communication security. A parallel literature studies the security of Android's *control* layer at the intent level. ComDroid [42] and Epicc [43] statically analyse inter-component communication (ICC) to identify intent-hijacking and ICC-mediated vulnerabilities, and XManDroid [44] mitigates privilege-escalation attacks that arise from uncontrolled inter-app collusion. These systems target individual IPC vulnerabilities rather than the SDK-mediated control plane AppScopes addresses, and they pre-date the migration of cross-app authority from OS permissions into SDK-internal credentials — the gap that motivates our work.

In contrast to prior work, which either *analyses* individual mechanisms (taint tracking, ICC static analysis) or *isolates* at coarse granularity (apps, containers, libraries), AppScopes introduces fine-grained, function-driven enforcement over the cross-app interaction graph itself. This complements existing techniques by providing an enforceable control layer over SDK-mediated collaboration rather than analysing or restructuring individual applications.

X. Conclusion

We studied Android's app-collaboration layer by decomposing it into *discovery* (which apps could be identified) and *control* (how apps could act on others). We measured 9,539 popular Play Store apps and found widespread overbroad discovery: many apps listed large, unrelated target sets via `<queries>`, enabling high-fidelity device fingerprinting. Our analysis of collaboration SDKs showed coarse-grained, credential-based privilege models that lacked per-target access control.

Motivated by these findings, we designed AppScopes and introduced *collaboration scopes* as a first-class primitive for governing cross-app interactions. We implemented a Knox-based prototype and evaluated it across three device generations. AppScopes shows minimal latency overhead ($\approx$30 µs per call on recent hardware, $\approx$0.2 ms on older devices) and negligible memory footprint ($\approx$28 bytes per wrapper). Our results demonstrate the practicality of enforcing function-driven, fine-grained access control for SDK-mediated collaboration. Full, mandatory enforcement would require integrating scope checks into the Android framework, but AppScopes provides a concrete, enforceable control layer and a two-layer decomposition of the collaboration surface to guide future platform changes.

AI Disclosure

We used Gen AI tools (such as Claude and Gemini) to assist with developing our research prototypes. The tools developed were instrumentally used to perform data collection and analysis (in Section III), implementation, and evaluation of our access control mechanism (in Section VII). The authors verified the correctness of all tools by manually cross-checking all produced results. We also used Claude to help polish the paper draft. Authors prepared the initial draft of the entire paper and minimally used GenAI tools to polish the writing. Any generated content was carefully reviewed by the authors for accuracy.

References

[1] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transac-*

tions on Privacy and Security (TOPS), vol. 24, no. 3, pp. 1–35, 2021.

[2] Google Play Console Help, "Use of the broad package (app) visibility (query_all_packages) permission," https://support.google.com/googleplay/android-developer/answer/10158779, 2021, accessed: 2026-04-21.

[3] Android Developers, "Package visibility filtering on android," https://developer.android.com/training/package-visibility, 2026, last updated 2026-03-30. [Online]. Available: https://developer.android.com/training/package-visibility

[4] Peabee, "Everyone knows what apps you use," https://peabee.substack.com/p/everyone-knows-what-apps-you-use, 2024, accessed: 2026-04-21.

[5] H. Lu, Y. Liu, X. Liao, and L. Xing, "Towards privacy-preserving social-media sdks on android," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 647–664.

[6] Samsung Knox, "Mobile security threats to enterprise devices in 2026," https://www.samsungknox.com/en/blog/mobile-security-threats-to-enterprise-devices-in-2026, 2026, accessed: 2026-04-21.

[7] Cybersecurity Dive, "Cisa urges organizations to harden endpoint security following stryker attack," https://www.cybersecuritydive.com/news/cisa-organizations-harden-endpoint-security-stryker-attack/815193/, 2026, accessed: 2026-04-21.

[8] Microsoft Defender Security Research Team, "Intent redirection vulnerability in third-party sdk exposed millions of android wallets to potential risk," https://www.microsoft.com/en-us/security/blog/2026/04/09/intent-redirection-vulnerability-third-party-sdk-android/, April 2026, accessed: 2026-04-21.

[9] Android Open Source Project, "Android security overview," https://source.android.com/docs/security/overview, 2008, accessed: 2026-04-19.

[10] W. Enck, M. Ongtang, and P. McDaniel, "A study of android application security," in *USENIX Security Symposium*, 2011.

[11] Android Developers, "Sdk runtime in privacy sandbox on android," https://developer.android.com/design-for-safety/privacy-sandbox/sdk-runtime, 2023, accessed: 2026-04-19.

[12] UK Competition and Markets Authority, "Google privacy sandbox investigation," https://www.gov.uk/cma-cases/investigation-into-googles-privacy-sandbox-browser-changes, 2023, accessed: 2026-04-19.

[13] J. Gagel, K. Heid, and J. Heider, "Permission granted? how android's app list protection fails in practice," in *European Symposium on Research in Computer Security*. Springer, 2025, pp. 544–557.

[14] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[15] Samsung, "Samsung knox sdk overview," https://docs.samsungknox.com/dev/knox-sdk/, 2024, accessed: 2026-04-19.

[16] Meta Platforms, "Facebook sdk for android documentation," https://developers.facebook.com/docs/android, 2024, accessed: 2026-04-19.

[17] Google, "Sign in with google for android," https://developers.google.com/identity/sign-in/android, 2024, accessed: 2026-04-19.

[18] ——, "Google mobile ads sdk," https://developers.google.com/admob/android/quick-start, 2024, accessed: 2026-04-19.

[19] ——, "Firebase analytics for android," https://firebase.google.com/docs/analytics, 2024, accessed: 2026-04-19.

[20] Android Developers, "Health connect overview," https://developer.android.com/health-and-fitness/guides/health-connect, 2023, accessed: 2026-04-19.

[21] The Register, "Facebook sues developer over data-harvesting sdk used in mobile apps," https://www.theregister.com/2020/02/28/facebook_sues_developer/, 2020, accessed: 2026-04-21.

[22] M. Grace, Y. Zhou, Z. W. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *ACM WiSec*, 2012.

[23] Android Developers, "Intents and intent filters," https://developer.android.com/guide/components/intents-filters, 2024, accessed: 2026-04-21.

[24] H. Kawabata, T. Isohara, K. Takemori, A. Kubota, J. Kani, H. Agematsu, and M. Nishigaki, "Sanadbox: Sandboxing third party advertising libraries in a mobile application," in *2013 IEEE International Conference on Communications (ICC)*. IEEE, 2013, pp. 2150–2154.

[25] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 9–18.

[26] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 553–567.

[27] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "Flexdroid: Enforcing in-app privilege separation in android." in *NDSS*, 2016.

[28] F. Wang, Y. Zhang, K. Wang, P. Liu, and W. Wang, "Stay in your cage! a sound sandbox for third-party libraries on android," in *European symposium on research in computer security*. Springer, 2016, pp. 458–476.

[29] Y. Zhang, M. Yang, G. Gu, and H. Chen, "Finedroid: Enforcing permissions with system-wide application execution context," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 3–22.

[30] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 691–706.

[31] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: Supporting and enforcing security profiles on smartphones," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 3, pp. 211–223, 2014.

[32] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard–enforcing user requirements on android apps," in *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 543–548.

[33] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun,

L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[34] S. Smalley and R. Craig, "Security enhanced (se) android: bringing flexible mac to android." in *Ndss*, vol. 310, 2013, pp. 20–38.

[35] L. K. Yan and H. Yin, "{DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis," in *21st USENIX security symposium (USENIX security 12)*, 2012, pp. 569–584.

[36] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.

[37] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 814–828, 2018.

[38] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[39] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[40] J. P. Achara, G. Acs, and C. Castelluccia, "On the unicity of smartphone applications," in *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*, 2015, pp. 27–36.

[41] A. Kurtz, H. Gascon, T. Becker, K. Rieck, and F. Freiling, "Fingerprinting mobile devices using personalized configurations," *Proceedings on Privacy Enhancing Technologies*, 2016.

[42] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011, pp. 239–252.

[43] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 543–558.

[44] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.