

Argus: Detecting Access Control Vulnerabilities in Android Framework using Large Language Models

by

Haseeb-Ur-Rehman Faheem

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Haseeb-Ur-Rehman Faheem 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Access control (AC) inconsistencies in Android’s framework APIs are a persistent security challenge, especially in customized ROMs. While convergence-based and probabilistic tools have made progress in detecting these inconsistencies, they suffer from either high false positive rates or limited coverage due to reliance on domain-knowledge and manually defined inference rule. Large Language Model (LLMs) offer promise but tend to overprotect APIs when operating without sufficient Android specific context.

In this work, we present ARGUS, a novel hybrid LLM-driven pipeline that combines static code analysis, code embeddings, and context-aware prompting to improve AC recommendations for Android APIs. ARGUS begins by extracting path-sensitive summarized program paths from decompiled ROMs and uses rule-based prompting to guide LLMs in identifying focal functionality (i.e., security-relevant sinks). These sinks, along with their known AC, are embedded and stored in a code embedding. Given a target API, ARGUS’s retrieval system queries this database to identify similar functionalities with known AC levels. These aligned examples are then incorporated into a second, detailed prompt that instructs the LLM to consider relevance, sensitivity rankings, and Android-specific context; enabling precise, context-aware AC recommendations.

We evaluate ARGUS on five custom ROMs and demonstrate its effectiveness in uncovering previously undocumented AC vulnerabilities. Among these are unauthorized camera parameter modification and spoofing of system crash reports (Vivo Y33s), exposure of SIM lock state (Vivo Y33s), unprotected refresh rate customization and retrieval of all package names (OnePlus 12), and unrestricted access to Wi-Fi hotspot management APIs, such as setting client blocklists (Infinix Smart 8). Several findings have been acknowledged by vendors, highlighting ARGUS’s practical impact.

Acknowledgements

I would like to thank my supervisor, Dr. Yousra Aafer, for her guidance and support.

Special thanks to Mama, Papa, and Muneeb for their unwavering support and belief in me throughout this journey. I am also deeply grateful to Karim Kati for his constant help and encouragement, especially during tough times, may you achieve great success in all your endeavors. I would also like to give an honorary mention to Parjanya Vyas for his generous help and support along the way.

Table of Contents

Author’s Declaration	ii
Abstract	iii
Acknowledgements	iv
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Background	3
3 Motivation	5
3.1 Current Access Control Recommendation Systems	5
3.2 LLMs as a promising solution	6
3.3 Limitations of Access Control Recommendation using LLMs without context	6
3.4 Limitations of Deriving Rules from Unprotected APIs	8
3.5 Context-Aware API Retrieval via Embedding Search for LLM-Based AC Recommendation	9
4 Approach	11

5	Design	13
5.1	Static Analysis to Extract Summarized Program Paths	13
5.2	Sink Extraction via LLMs	15
5.3	Sink Embedding and Semantic Storage	17
5.4	API Alignment via Similar Sink Retrieval	17
5.5	Context-Aware AC Recommendation	18
6	Evaluation	20
6.1	EVALUATION	20
6.2	What are the limitations of using LLMs in AC recommendation when operating without Android-specific context?	21
6.3	Which large language model achieves higher accuracy using ARGUS on the AOSP dataset?	22
6.4	What is the effect of the similarity score on coverage and accuracy?	24
6.5	Can Argus detect AC inconsistencies in custom ROMs?	26
6.6	How does ARGUS compare to previous approaches?	29
	6.6.1 Convergence-Based Approach	29
	6.6.2 Probabilistic Approach	30
7	Case Studies	33
7.1	Unauthorized Refresh Rate Modification in OnePlus 12	33
7.2	Exposure of Sensitive SIM and Device Lock Information in Vivo	34
8	Related Works	35
8.1	Vulnerability Detection Using LLMs	35
8.2	Inconsistent Access Control Detection in Android	36
9	Limitations and Future Work	38
9.1	Diverse Access Control Checks	38
9.2	Selecting the Longest Path for Focal Functionality Identification	39

10 Conclusion	40
References	41

List of Figures

3.1	Example of incorrect LLM access control recommendation.	7
4.1	System Diagram	11
5.1	Prompt template for extracting Focal Functionality in Android API.	16
5.2	Prompt template for assigning AC to Android APIs.	18
6.1	None-None prediction accuracy	25
6.2	Categorization of Positives Reported by ARGUS	27

List of Tables

6.1	LLM Predicted AC Levels for AOSP APIs (Single-API Analysis)	21
6.2	Comparison of Access Control Prediction Accuracy by Model (Threshold: 0.70)	24
6.3	Impact of Cut-off Criteria	24
6.4	Coverage of Custom APIs across ROMs	26
6.5	Summary of Discovered Vulnerabilities	28

Chapter 1

Introduction

AC inconsistencies in Android’s framework Application Programming Interfaces (APIs) are a persistent security concern, particularly in the context of customized ROMs, where vendor modifications may alter or weaken the AC for an API. These inconsistencies can enable unauthorized access to sensitive operations, introducing security vulnerabilities and exposing end users to potential threats.

There are 2 common approaches for evaluating AC inconsistencies in Android: convergence based analysis and probabilistic inference. Convergence based tools such as Kratos [18], Acedroid [1], and ACMiner [9] assess AC by identifying APIs that converge on a shared resource and comparing their AC enforcement levels. If one path enforces a lower AC than another, the API is flagged as inconsistent. However, Android APIs often contain auxiliary instructions (i.e., those unrelated to the main functionality) such as logging, state updates and debugging logic. This leads convergence based tools to incorrectly infer the target of AC, causing high false positive rate. To fix this, Poirot [5] adopts a probabilistic-based approach to infer likely main resources, by treating the resource-to-AC association as uncertain. While Poirot improves precision, it relies on domain-knowledge and manually defined inference rules, which limit its ability to capture other rules missed by the rich semantic and data flow hints in ROMs. Its performance is further constrained by sparse observations, which leads to a very low coverage (around 60%).

Recently, LLMs have emerged as a powerful tool for code analysis and vulnerability detection [6, 15, 21, 16, 14, 22]. These models demonstrate the ability to understand complex control and data flows [21, 15, 14], reason effectively about API semantics [16, 14], and detect security flaws in ways that generalize beyond pattern matching [6, 15, 22]. Empirical evaluations confirm that LLMs can prune false positives [14], uncover real-world

bugs [22], and perform semantic reasoning even over obfuscated or incomplete code [6], supporting their promise in automated code understanding and robust security auditing.

Motivated by this, we hypothesize that LLMs can improve the task of AC recommendation in Android framework. However, our initial experiments show that when used in isolation, LLMs tend to overprotect APIs, hence limiting functionality. Without sufficient Android-specific context, the models flag non-sensitive operations as potentially requiring AC. For example, LLMs may incorrectly recommend a system-level protection for an API that checks the lock state of the device, citing “a locking operation” as justification. Note that in the Android context, reading the device lock state is considered a non-sensitive operation and does not typically require an AC, unlike writing or modifying lock settings, which are more privileged. This results in an excessive permission enforcement that undermines usability and violates the principle of least privilege.

To overcome this, we propose a hybrid, LLM-based AC recommendation system for APIs. Our approach features a synergy of static code analysis and code embedding, with LLMs. Our approach begins by extracting a summarized execution path from the API bytecode and prompting the LLM to identify potential sinks—key instructions that capture the API’s core functionality. We then embed these sinks and use a retrieval system to find other APIs in the ROM that exhibit similar sink behavior and have a *known correct* AC level. These retrieved $\langle \textit{aligned functionality} - \textit{correct AC} \rangle$ examples are provided to the LLM in a second prompt, enabling it to make an informed and context-aware AC recommendation for the target API. Our experimentation demonstrates the following key findings:

- **LLMs without Android context lead to overprotection:** When used in isolation, LLMs are unable to distinguish between benign and sensitive operations, resulting in consistently overprotective AC recommendations.
- **Prompting with contextual examples improves accuracy and coverage:** By guiding LLMs with Android-specific rules and retrieved examples, our approach achieves accuracy of 84.5% on AOSP version 14 and 84.9% on AOSP version 15, with coverage rates of 90.1% and 90.7%, respectively, significantly surpassing traditional inconsistency detection tools.
- **Real-world impact:** We evaluate ARGUS on five custom ROMs—OnePlus 12, Vivo Y33s, Samsung A16, Amazon Tab, and Infinix Smart 8—identifying 32 previously undocumented AC inconsistencies. We successfully created proof-of-concept (PoC) exploits for 28 of these cases, demonstrating significant security implications, many of which have been acknowledged by the respective vendors.

Chapter 2

Background

AC in Android is crucial for protecting sensitive device operations and user data. AC enforcement in Android is categorized into four levels as originally proposed by Acedroid [1]: None, Normal, Dangerous, and System. Each level corresponds to varying degrees of sensitivity and risk associated with the protected resources:

- **None:** Requires no AC checks and is available to all applications.
- **Normal:** Minimal risk permissions automatically granted at install-time without explicit user consent. Examples include `ACCESS_NETWORK_STATE` and `ACCESS_WIFI_STATE`.
- **Dangerous:** Grants access to sensitive user data or critical device features, requiring explicit runtime user approval. Typical permissions in this category include `READ_CONTACTS`, `WRITE_CONTACTS`, and `GET_LOCATION`.
- **System:** Reserved for highly privileged operations accessible exclusively to system-level applications. An example is the permission `SEND_RESPOND_VIA_MESSAGE`.

These AC levels are enforced through various mechanisms, including application-specific checks such as permission verification and package name validations, as well as user-specific checks, such as user ID (UID) verification.

Customized Android ROMs frequently integrate vendor-specific features and services to enhance user experience and device functionality. Vendors, such as Vivo, OnePlus, and Infinix often introduce custom services and APIs, for instance, Vivo adds services to introduce custom SIM-related functionalities, OnePlus implements APIs for screen refresh

rate management, and Infinix includes APIs for Wi-Fi hotspot management. These vendors significantly augment their ROMs, resulting in extensive numbers of custom APIs: 962 in OnePlus 12, 849 in Vivo, and 2971 in Samsung A16, which highlights the extensive scale and complexity of vendor modifications. However, due to the absence of precise and formal security specifications, these vendor specific additions frequently lead to uncoordinated and inconsistent implementations of AC. Such inaccuracies in AC enforcement can inadvertently expose sensitive resources or actions, allowing unauthorized applications access to critical operations, which poses significant security risks.

These inconsistencies primarily arise from two main issues. First, vendors often implement AC checks inconsistently across different APIs and framework layers (Java and native code), leading to varying enforcement levels for similar resources. Second, the subjective nature of developers' sensitivity assessments, driven by a lack of formal AC specifications, introduces ambiguity and uncertainty into security implementations. Consequently, vulnerabilities emerge when incorrect or insufficient permissions are assigned, highlighting the critical need for precise and context-aware AC recommendation systems to mitigate potential security breaches in Android framework.

Chapter 3

Motivation

3.1 Current Access Control Recommendation Systems

Prior tools like Kratos [18], Acedroid [1] and ACMiner [9] rely on convergence analysis to assess the adequacy of AC. These tools analyze different paths in a ROM that lead to a shared convergence point. If one path enforces a lower AC than the other, it is flagged as an inconsistency. However, since the APIs contain a lot of auxiliary code, such as logging, debugging statements, or internal state updates, the convergence point might not be the intended target of AC. Therefore, these tools suffer from a high false positive rate.

To suppress false positives, recent work introduces Poirot [5], a probabilistic-inference based tool that tries to pinpoint the target of AC. Unlike prior tools that assume a resource control- dependent on an AC is its intended target, Poirot treats this association as uncertain and resolves it using probabilistic inference. It performs path-sensitive static analysis to collect basic AC facts and assigns confidence scores to potential resource-protection pairs. Poirot then propagates protections using implication constraints based on reachability, triggering conditions, name correlation and other structural and semantic hints. These facts are then aggregated by a probabilistic engine to generate high-confidence protection recommendations, enabling Poirot to suppress false positives and uncover implicit inconsistencies that prior rule-based tools often miss. Unfortunately, Poirot suffers from multiple limitations. Since the inference rules and the preset probability values are manually defined, the tool can miss a lot of semantics in the ROM, which is rich in structural, semantic, and data-flow hints. Moreover, slight variance in these preset values can noticeably impact the accuracy of the predictions. Additionally, certain resources may lack sufficient observations across the ROM, preventing Poirot from confidently recommending AC for

them. This limitation is reflected in the tool’s relatively low coverage (around 60%), where coverage refers to the proportion of APIs for which the tool can successfully provide an AC recommendation.

3.2 LLMs as a promising solution

LLMs have recently emerged as a promising solution for advanced code analysis, effectively addressing key limitations of traditional static analysis in security contexts. These models exhibit an impressive capability to understand complex control-flow and data-flow patterns, enabling them to spot subtle bugs and even sensitive operations that might not be caught by conventional convergence based or probabilistic tools. For instance, integrating LLM reasoning into static analysis pipelines has been shown to prune many false positives while validating true issues, even uncovering new real-world kernel vulnerabilities in large codebases [14]. Similarly, autonomous LLM-driven auditors can traverse entire repositories, leveraging agent-based memory and data-flow tracking to pinpoint true bugs efficiently with minimal overhead [22]. Overall, empirical evaluations confirm that state-of-the-art LLMs are capable of performing sophisticated code auditing and semantic reasoning across code bases (in some cases, even handling slightly obfuscated or incomplete code) beyond simple pattern matching [6].

We hypothesize that LLMs can improve the task of AC recommendation for APIs performing *a sensitive operation*. However, due to lack of sufficient Android context, they might be unable to discern other less sensitive operations. The latter requires (1) understanding intricacies of Android-specific resource to AC associations, and (2) collectively analyzing other APIs to gather relevant contextual insights, such as similar fields that are accessed or modified, objects instantiated, invoked native methods, and the data flow involving parameters, with their assigned AC. To verify our hypothesis, we performed two experiments using AOSP codebase, which is often considered ground truth in Android literature [1, 3, 5].

3.3 Limitations of Access Control Recommendation using LLMs without context

In our first experiment, we strip AC implementations from AOSP APIs and prompt Claude Sonnet 3.7 to generate an AC recommendation. For input, we generate summarized

System Prompt: You are an expert in Android Framework Security. Determine the appropriate access control required for the following API.

User Prompt (Simplified API Code):

```
1 @Override // android.app.trust.ITrustManager
2 public boolean isDeviceLocked(int userId, int deviceId){
3     Log.d("API", "Checking_user_and_device_state");
4     int callingUid = ITrustManager.Stub.getCallingUid();
5     if (isVirtualDevice(callingUid, deviceId)) {
6         return false;
7     }
8     int userHandle = ActivityManager.handleUser(callingUid, userId);
9     return mDeviceLockedForUser.get(userId, true);
10 }
```

LLM Response: *The method logs device states and interacts with critical system entities, potentially exposing crash data or sensitive states. Thus, it should require SYS_OR_SIG protection.*

Figure 3.1: Example of incorrect LLM access control recommendation.

program paths for each APIs (details discussed later). We found that the model exhibited substantial hallucinations, reporting overly restrictive AC recommendations. Particularly, it mistakenly recommended that the majority of non-sensitive APIs (i.e., those not enforcing an AC check in AOSP) should require a *SYSTEM* level permission. A detailed breakdown of the reports can be found in Section 6.2. When asked for justification, the model often interpreted any form of system interaction, such as logging internal state, checking UID, or invoking internal methods with sensitive keywords such as locks, passwords, etc, as potentially sensitive – which is incorrect under the Android context. Figure 3.1 illustrates an example when the model incorrectly recommends a *SYSTEM* permission for an API that simply checks the device lock status. The LLM justifies high level of AC by citing the interaction with system components, without accounting for the fact that such operations do not leak sensitive state to the third-party applications. This highlights a broader limitation that current LLMs lack the Android context and struggle to accurately assess sensitivity when operating on a single API in isolation. This limitation, i.e., overprotective AC recommendation, limits usability and violates the principle of least privilege.

3.4 Limitations of Deriving Rules from Unprotected APIs

In the second experiment, we attempted to improve the prompts by providing concrete rules corresponding to non-sensitive operations. To this end, we use Claude Sonnet 3.7 to derive such rules. Specifically, we compiled a list of AOSP APIs that do not require AC and prompted the LLM to generate rules explaining why the APIs had no AC. We found that the generated rules were often incorrect:

Generated Rules from Claude 3.7

- **APIs that perform read-only operations, logging, or non-mutative checks may not require stringent access control since they inherently pose less risk.**
- **Ensure that APIs carry out preliminary security checks and extensive logging ('log'). This indirect form of control can provide sufficient safeguards against unauthorized access.**
- **APIs which adapt behavior based on context or environment can rely on these conditions as a form of access safety, reducing the necessity for additional access controls.**

While these rules may appear plausible at first glance, they are problematic under the Android security model:

- The first rule assumes that read-only operations are inherently low-risk. However, even read-only APIs can leak sensitive information. For instance, APIs like `getInstalledApplications` reveal installed apps on the device and are protected in AOSP, despite being non-mutative.
- The second rule suggests that logging or preliminary checks can serve as effective substitutes for AC. This is incorrect, since logging may record internal states for debugging, but it does not prevent unauthorized access. Some APIs perform logging as a side effect but still require AC to prevent exposure of privileged behavior.
- The third rule assumes that context-aware logic, such as checking if a user or app exists, is sufficient to eliminate the need for explicit protection. This overlooks the fact that such checks do not enforce AC on their own. For example,

`deleteApplicationCacheFilesAsUser` performs validation checks for application and user, but is still protected in AOSP, highlighting the insufficiency of relying solely on environmental conditions.

Clearly, this method of deriving general rules from unprotected APIs was unsuccessful, as the generated rules did not reflect accurate or secure access control practices. Detailed results of these failed LLM experiments can be found in Section 6.2.

3.5 Context-Aware API Retrieval via Embedding Search for LLM-Based AC Recommendation

Despite the limitations above, we observe that LLMs can make *correct* AC recommendation if sufficient Android context is provided. Specifically, we note two crucial points for AC recommendation: (1) focal functionality identification and (2) API functionality alignment for AC inference. Although existing AC tools resort to convergence analysis [1, 18, 9] or probabilistic analysis [5] to realize these capabilities, LLMs can leverage their code analysis ability along with extensive contextual understanding using carefully structured, Android-specific rules to effectively address these challenges.

Focal Functionality Identification: As noted by prior work [1, 18, 9], pinpointing the main functionality(s) of an API is critical for AC recommendation. Given an API, an AC analysis system identifies resources¹, relevant to the API’s enforced AC. Based on Axplorer’s taxonomy [3], Android resources may correspond to three primitive types: field access and update, internal method invocation (e.g., native methods, file access methods, non-exposed service methods) and API method invocation (an API may invoke another API). Such resources form sensitive code points that require protection to prevent unauthorized access. For example, in Figure 3.1, line 9 accesses the field `mDeviceLockedForUser` using the parameter `userId`. This is a field access related to the user state and indicates a resource. Our initial experiments show that LLMs can successfully pinpoint these resources when provided with carefully structured Android-specific rules describing sensitive resources, and also differentiate between auxiliary or benign code. (discussed in Section 4).

Functionality Alignment for AC Inference: Due to the lack of (complete) security specifications, prior works resort to differential and consistency analysis to assess the validity of an API’s AC. Essentially, the works *align* the functionality of a given API with others in order to compare their enforced AC. Comparison may include multiple APIs with varying

¹as termed by Poirot

AC, and hence often requires further reasoning to infer the correct AC. We observe that LLMs are able to derive the proper AC in such cases.

Chapter 4

Approach

In lights on these observations, we propose ARGUS, an LLM-driven analysis pipeline for AC recommendation in Android. To facilitate analysis, ARGUS features static code analysis and code embedding. At a high level, ARGUS proceeds as follows: ARGUS preprocesses an API's implementation through static code analysis to facilitate analysis by the LLM. To leverage the LLM's inherent code-analysis ability, ARGUS generates prompts for focal functionality identification and for AC recommendation. To effectively compare and align the functionality of different APIs, the identified focal instructions are converted into vector embeddings, which capture both, semantic and syntactic features of the code. These embeddings are then used to retrieve semantically similar APIs with known AC levels, which are incorporated into a second, context-rich prompt to guide the LLM's final recommendation.

Figure 4.1 illustrates the analysis stages of ARGUS.

(1) Static Analysis: In the initial phase, given an API in bytecode format, ARGUS performs path-sensitive analysis to generate its execution paths, summarizing implemented

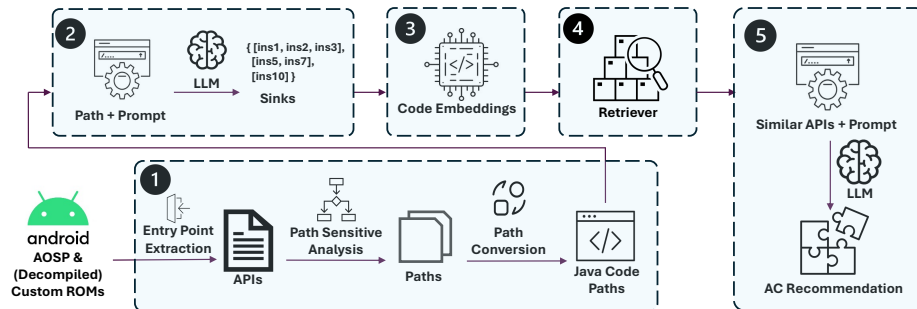


Figure 4.1: System Diagram

functionality. The paths are then abstracted into Java-like code (to facilitate upcoming LLM code analysis). This design choice enables for a token efficient and interpretable format, enabling effective reasoning.

(2) Sink Extraction: In the second phase, ARGUS prompts the LLM to analyze each API’s abstracted execution paths to identify instructions corresponding to focal functionality. The prompts include Android domain-specific context in the form of positive and negative rules. The responses consist of ranked focal instructions for each API that constitute a sink and are accompanied by chain-of-thought reasoning to minimize hallucinations. The latter are also useful for (1) improving the accuracy and performance of the LLM analysis, and (2) providing security experts with insights into the LLM’s reasoning process for better assessment.

(3) Sink Embedding and Storage: In the third phase, the identified sinks are transformed into vector embeddings using CodeT5+ model. The embeddings, which capture both semantic and syntactic properties of the code, facilitate upcoming API functionality alignment. These embeddings, along with their associated known AC levels, are saved in a dedicated database for future reference.

(4) Functionality Alignment for AC inference: Given a target API, ARGUS’s retrieval system queries the database to identify aligning functionality with a known AC. To this end, ARGUS relies on semantic similarity analysis and prioritizes results based on sensitivity and domain-specific rules (e.g., ROM origin).

(5) Context-Aware AC Recommendation: Finally, using the retrieved examples of similar functionalities and their known AC levels (*<aligned functionality - correct AC >*), ARGUS generates a second detailed prompt for the LLM including Android-specific context. The detailed prompt instructs the LLM to evaluate the relevance and sensitivity rankings of matched sinks. This structured and context-rich approach helps ensure precise AC recommendations.

Chapter 5

Design

In this section, we describe the design of the ARGUS.

5.1 Static Analysis to Extract Summarized Program Paths

To accurately extract these APIs, ARGUS first identifies system services that are registered through methods such as `ServiceManager.addService()` or `ServiceManager.publishBinderService()`. It statically analyzes the framework’s bytecode by constructing call graphs for all declared methods. ARGUS then traverses these call graphs to pinpoint invocation sites of the aforementioned registration methods. Utilizing the def-use chain, ARGUS resolves method arguments to determine the registered service classes. Each identified service class implements an interface containing publicly exposed APIs. Through Class Hierarchy Analysis (CHA), ARGUS examines these interfaces to collect all public API entry points, ultimately generating a comprehensive service-to-API mapping for the analyzed Android image. For each API, an inter-procedural control flow graph (ICFG) is generated to extract path-sensitive execution traces. These paths represent potential data and control flows within the API.

Feeding all execution paths to an LLM is costly and counterproductive. Many execution paths involve early returns resulting from failed input validations or prior AC checks. These paths do not contain the focal functionality (security-sensitive operations) of the API and only clutter the input to the LLM. Including such unnecessary paths bloats the prompt,

increases the inference cost, and often confuses the model, potentially exceeding its context window.

For instance, consider the APIs shown in Listing 5.1:

Listing 5.1: Example APIs illustrating input validation paths

```
@Override // android.media.IAudioService
public void setMultiAudioFocusEnabled(boolean z) {
    if (mediaFocusControl == null ||
        mediaFocusControl.getMultiAudioFocusEnabled() == z) {
        return;
    }
    this.mMediaFocusControl.updateMultiAudioFocus(z);
}

@Override // android.hardware.input.IInputManager
public int getkeyCodeForKeyLocation(int i, int i2) {
    if (i2 <= 0 || i2 > KeyEvent.getMaxKeyCode()) {
        return 0;
    }
    return this.mNative.getKeyCodeForKeyLocation(i, i2);
}
```

In these examples, the input validation paths alone cause the LLM to miss critical focal operations such as `updateMultiAudioFocus(z)` and `getKeyCodeForKeyLocation(i, i2)`, significantly limiting its ability to accurately assess the APIs' main functionality.

Thus, selectively filtering paths to exclude these early return scenarios helps maintain the clarity for the LLM and efficiency of the prompts, which enhances the accuracy of AC recommendations.

Path Filtering: Therefore, ARGUS filters paths using two lightweight rules:

Failed Input Validation Paths A path is considered to have failed input validation if it contains a conditional branch where one of the conditions involves an API parameter, and the rest of the path only consists of debug-related instructions such as log/slog statements, its data-dependent instructions, and a return statement.

Failed AC Paths Similarly, a path is identified as failing an AC check if, following an AC statement, it contains only log/slog instructions, their data-dependent instructions, and return statements.

Paths matching either rule are removed to reduce noise, minimize prompt size, and help the LLM focus on security-relevant operations.

Conversion to Java-like Code: Filtered paths are abstracted into a Java-like representation. Converting the paths to Java-like code improves token efficiency, readability and semantic clarity, making it easier for the LLM to understand the API’s behavior and reason about the functionality. ARGUS resolves method calls, data flows, and field accesses/updates, and generates code at three depths (1–3), with a cap to avoid path explosion. The conversion includes handling various instruction types, a few are listed below:

- **Invoke Instruction:** The method name, parameters, and fields involved (for dynamic methods) are identified. For non-void methods, the return type is resolved. Internal method calls are assigned to variables corresponding to their return types.
- **Put Instruction:** The class name, field name, and assigned values are resolved and represented clearly.

Java-like code is generated at multiple levels of depth:

- Depth = 1: Only the top-level method, without exploring any inner methods.
- Depth = 2: The top-level method and one inner method.
- Depth = 3: The top-level method and two inner methods.

This step produces numerous summarized program paths for further analysis.

5.2 Sink Extraction via LLMs

To identify an API’s focal functionality that may require protection, we query the LLM with a carefully structured prompt.

Prompt Design A detailed prompt is prepared to instruct the LLM clearly about identifying focal functionality within the API. The LLM is first assigned the role of an expert in Android security, particularly focused on AC for Android APIs (Figure 5.1).

Then, the prompt provides explicit positive and negative rules to guide the LLM in identifying potential sinks:

Positive Rules Examples:

- Method invocations similar in naming to the API or using API parameters.

Role: You are an expert in Android Framework Security. Task: Determine if an Android API contains sinks requiring access control. Rank the sinks from most sensitive to least sensitive. Hints: Clearly apply the provided rules to identify sinks: Positive Rules • Identify method invocations, field accesses/modifications, and object creations that propagate sensitive data. • Trace explicit dataflows from API parameters to returned values or internal state. Negative Rules: • Exclude auxiliary operations such as locking mechanisms,. • Exclude generic utilities, logging, and input validation methods. Example: User: [API Name & Java Code] System: <i><Potential Sink Information: lines X, Y, Z></i> <i><Data-flow facts: method invocation @lineX $\hookrightarrow$ object creation @lineY $\hookrightarrow$ return @lineZ></i> Question: Given API [API_NAME], what instructions constitute sinks?

Figure 5.1: Prompt template for extracting Focal Functionality in Android API.

- Field accesses or updates similar in naming to the API or involving API parameters.
- Creation of new objects related in naming or data-flow to the API parameters.

Negative Rules Examples:

- Utility methods like `valueOf`, `toString`, or `isEmpty`.
- Logging or validation methods such as `log` or sanity checks.
- Auxiliary operations like obtaining context or acquiring locks.

Additionally, the prompt includes an example API named *getLightState*, clearly demonstrating how data flows through the API, internal field interactions, and method invocations

contribute to focal functionality identification. The example concludes with a structured JSON output demonstrating the sinks identified, their corresponding instructions and detailed reasoning. The LLM is asked explicitly to provide reasoning in a step-by-step manner (chain-of-thought reasoning), and to generate a similar structured JSON output ranking the sinks from most to least sensitive.

Selecting Paths for LLMs The longest code paths at each depth level (1, 2, and 3) are selected and provided to the LLM. The longest paths contain the most detailed and comprehensive information about method invocations and potential sinks, maximizing the likelihood that critical security-sensitive operations, and hence focal functionalities, are captured for analysis by the LLM.

5.3 Sink Embedding and Semantic Storage

After running the prompt, the sinks identified by the LLM are stored using code embeddings, specifically employing the CodeT5+ model (other models like CodeBERT could also be used). Code embedding models convert code snippets into vector representations, capturing both syntactic and semantic information. These embeddings facilitate efficient similarity searches, allowing the system to detect related APIs effectively based on how closely they match at both the semantic and syntactic levels.

5.4 API Alignment via Similar Sink Retrieval

To recommend AC for a test API, ARGUS uses a retriever to query its embedding database to identify APIs with similar focal instructions and known AC enforcement. Specifically, the embedding of each extracted sink in the target API is compared against the embeddings of sinks from other APIs in the ROM using cosine similarity:

$$\text{sim}(P_o, P_r) = \frac{\vec{P}_o \cdot \vec{P}_r}{\|\vec{P}_o\| \|\vec{P}_r\|}$$

where $\vec{P}_o$ and $\vec{P}_r$ denote the vector embeddings of the respective sink instructions. If the similarity between a sink in the test API and another sink in the ROM exceeds a predefined threshold, the corresponding API is considered a match. We evaluate the impact of different thresholds in Section 6.1. Each retained API contributes a tuple of the form: *<aligned functionality - correct AC>*, which is later used to guide LLM reasoning.

AOSP Prioritization. Since custom ROMs may themselves contain AC vulnerabilities, ARGUS prioritizes matches from AOSP when selecting similar APIs. Specifically, for each test API, the system attempts to retrieve four matching APIs above the similarity threshold. If fewer than four high-confidence matches are found within AOSP, additional examples are drawn from protected APIs (**NORMAL**, **DANGEROUS**, or **SYS**) in the custom ROM. This preference helps ensure that the retrieved examples reflect known-good AC enforcement practices and provide more reliable context to the LLM.

5.5 Context-Aware AC Recommendation

Role: You are an expert in Android Framework Security. Task: Assign Access Control levels to Android APIs using provided ground truths of similar APIs. Guidelines: Follow the priority below clearly: • Method and Service Context (Highest Priority): APIs with counterpart actions (e.g., createX/deleteX, grantX/revokeX) must have matching access levels. • Sink Sensitivity (Secondary Priority): Prefer exact sensitivity matches. • Setter vs. Getter: Setters usually require stricter control than getters; treat separately from symmetric action pairs. Example: User: [Original API, Similar APIs & Ground Truth] System: <i><Analysis: Matching counterpart found; sink sensitivity comparison; setter/getter decision></i> Question: Given API [original_method], determine the correct access control level.
--

Figure 5.2: Prompt template for assigning AC to Android APIs.

Finally, ARGUS prompts the LLM again, this time for AC recommendation, using the test API and its aligned examples.

Prompt Structure. The prompt, as illustrated in figure 5.2 includes:

- The test API’s focal sinks and their sensitivity ranks.

- The top 4 most similar APIs, with their sinks, ranks, and known AC labels.
- Rules guiding how similarity, sink importance, and Android context should inform AC inference.

The prompt clearly states how the similarities between APIs and their sinks, as well as their respective AC levels, should be considered. For instance, if the highest-ranked sink in the test API matches the highest-ranked sink in a similar API, this information is highlighted explicitly. Lesser-ranked matches are also mentioned, as this information is critical to helping the LLM understand contextually relevant factors.

The LLM is instructed to choose from the four AC categories: **NONE**, **NORMAL**, **DANGEROUS**, or **SYS**, and provide justification. The output is structured in JSON format for consistency.

Chapter 6

Evaluation

We implement a prototype for ARGUS consisting of two main components: (1) a static analysis module and (2) an LLM module.

The static analysis component is built on top of WALA [20]. All static analysis experiments were conducted on an IBM Power LC922 server equipped with a 22-core CPU (2.6 GHz POWER9 processor) and 256 GB of main memory. This module processes the Android ROM, extracts inter-procedural control flow paths of the APIs, and generates Java-like code snippets representing execution paths for subsequent analysis.

For the LLM module, we utilize Ollama [17] to host open-source models locally. Specifically, we use Llama 3.3 (70B), DeepSeek-r1 (70B), and Gemma 3 (27B) models, running on NVIDIA A100 GPUs with 80 GB VRAM. The local models are quantized to 4-bit precision, which is the default quantization level provided by Ollama.

For the closed-source models, such as GPT and Claude variants, we interact directly with their respective online APIs.

6.1 EVALUATION

We design several experiments to assess ARGUS’s effectiveness and performance. Specifically, our evaluation aims to answer the following research questions:

1. **RQ1:** What are the limitations of using LLMs in AC recommendation when operating without Android-specific context?

2. **RQ2:** Which large language model achieves higher accuracy using ARGUS on the AOSP dataset?
3. **RQ3:** What is the effect of the similarity score on coverage and accuracy?
4. **RQ4:** Can ARGUS detect AC inconsistencies in custom ROMs?
5. **RQ5:** How does ARGUS compare to previous approaches?

6.2 What are the limitations of using LLMs in AC recommendation when operating without Android-specific context?

For the first experiment, we perform inter-procedural static analysis on the APIs in the Android ROM (AOSP v14) to obtain the summarized paths. For each API, we obtain up to 3 execution paths, each representing a depth from 1 to 3. These paths are then converted to java-like code snippets to prompt an LLM, specifically requesting recommendation for an AC.

We select a representative subset of APIs from different AC levels, including 300 APIs with no AC(NONE), 80 APIs categorized as NORMAL, 20 APIs requiring DANGEROUS, and 100 APIs as SYS_OR_SIG. The LLM is then prompted with these java-like code paths, as illustrated in Figure 3.1.

Table 6.1: LLM Predicted AC Levels for AOSP APIs (Single-API Analysis)

True AC Level	Predicted AC Level	Count	Percentage (%)
NONE (300)	SYS_OR_SIG	192	64.0
	DANGEROUS	60	20.0
	NORMAL	48	16.0
NORMAL (80)	SYS_OR_SIG	51	63.75
	DANGEROUS	23	28.75
	NORMAL	6	7.50
DANGEROUS (20)	DANGEROUS	12	60.0
	SYS_OR_SIG	8	40.0
SYS_OR_SIG (100)	SYS_OR_SIG	67	67.0
	DANGEROUS	28	28.0
	NORMAL	5	5.0

The majority of misclassifications are for APIs that required no permissions (None), yet the model incorrectly recommended System for 64% of these cases. Similarly, Normal APIs, which typically represent benign, user-level interactions and are automatically granted during installation of an application, are also frequently misclassified as requiring System permissions. This demonstrates a clear lack of contextual understanding by the LLM when provided with isolated code snippets.

For Dangerous and System APIs, the model also performs poorly, correctly identifying the protection level in only around 65% of the cases. This means that not only does the model overprotect None and Normal cases, but it also demonstrates significant inaccuracies in determining correct permissions for APIs that inherently require stricter security controls. Consequently, the model cannot be reliably trusted for precise AC recommendations without additional contextual information.

6.3 Which large language model achieves higher accuracy using Argus on the AOSP dataset?

Next, we evaluate the accuracy of different LLMs integrated within ARGUS, focusing on their ability to recommend AC levels for APIs extracted from AOSP versions v14 and v15. We perform inter-procedural static analysis to extract AC in the APIs, treating these implementations as ground truth following previous research.

For the experiment setup, we first remove AC implementations from the Java-like code snippets generated from control flow paths. We then prompt our system, ARGUS, to recommend appropriate AC levels using various LLMs. Table 6.2 presents the detailed prediction results categorized by the true AC labels (NONE, NORMAL, DANGEROUS, SYS_OR_SIG) across both AOSP versions v14 and v15.

Results: From the table, we observe distinct performance differences among the evaluated models: Claude 3.7 Sonnet, Claude 3.5 Haiku, LLama 3.3 (70B), DeepSeek-r1 (70B), Gemma 3 (27B), and GPT-4o-mini. Each row within the model categories provides detailed insights into prediction accuracy for each AC level.

- **Claude 3.7 Sonnet** performs the best for NONE APIs, achieving accuracies of 71.6% on v14 and 66.3% on v15. It also demonstrates strong performance for SYS_OR_SIG APIs, correctly identifying around 93% of these cases across both versions. However, the the model performs moderately for NORMAL APIs, with approximately 67%

accuracy, and poorly for DANGEROUS APIs, with lower accuracies around 57% and 67% on v14 and v15 respectively.

- **Claude 3.5 Haiku** shows a slightly lower performance compared to Claude 3.7 Sonnet, particularly for NONE APIs, with accuracies around 61-63%. It excels, however, at predicting SYS_OR_SIG APIs, achieving approximately 95% accuracy consistently. Its performance for NORMAL APIs is fair, around 62-63%, and slightly better for DANGEROUS APIs compared to Claude 3.7, around 71-73%.
- **LLama 3.3 (70B)** shows significant strength for NORMAL APIs, reaching an accuracy of 82.4% on v14 and 75.6% on v15. Its accuracy for NONE APIs is moderate, approximately 62%, while for DANGEROUS and SYS_OR_SIG categories, it maintains good performance around 71-73% and 92-93%, respectively.
- **DeepSeek-r1 (70B)** shows consistent and robust performance across categories, particularly excelling in SYS_OR_SIG APIs with over 91% accuracy. For NONE APIs, its accuracy ranges from approximately 65-71%, while achieving around 69-73% accuracy for both NORMAL and DANGEROUS APIs.
- **Gemma 3 (27B)**, being a smaller model, shows moderate accuracy for NONE APIs (around 59-63%) and perform well for NORMAL APIs, roughly around 67-71%. However, the accuracy significantly drops in correctly classifying DANGEROUS APIs, approximately 69-71%. Like the other models, Gemma excel in identifying SYS_OR_SIG APIs, consistently achieving above 92% accuracy.
- **gpt-4o-mini** shows relatively weaker performance across most categories. Its accuracy for NONE APIs is modest, at 58.2% for v14 and 52.6% for v15. It performs reasonably for NORMAL APIs, reaching 63.6% on v14 and 61.9% on v15. However, its performance drops for DANGEROUS APIs, with 68.4% on v14 and 72.7% on v15. For SYS_OR_SIG APIs, it maintains decent performance, around 90.7% and 92.8%.

Figure 6.1 illustrates the NONE→NONE prediction accuracy across different models and Android versions (v14 and v15). Claude 3.7 Sonnet achieves the highest accuracy for APIs labeled as NONE, with DeepSeek-r1 a close second. This finding is important because, as earlier experiments indicated, LLMs often mistakenly flag benign or non-sensitive operations as sensitive, causing high AC to be recommended. Since NONE-labeled APIs (APIs with no AC) represent safe actions that make up a substantial portion of the Android framework, accurately identifying these APIs is crucial for maintaining usability and adhering to the principle of least privilege. Given its superior accuracy in this category, we select Claude

Table 6.2: Comparison of Access Control Prediction Accuracy by Model (Threshold: 0.70)

Model	Label	NONE		NORMAL		DANGEROUS		SYS_OR_SIG	
		v14	v15	v14	v15	v14	v15	v14	v15
Claude 3.7 Sonnet	NONE	71.6	66.3	0.9	11.0	14.3	10.8	2.4	2.1
	NORMAL	10.1	12.4	66.7	68.5	15.2	11.0	2.6	2.4
	DANGEROUS	9.3	9.3	17.4	10.2	57.1	67.2	2.2	1.9
	SYS_OR_SIG	9.0	12.0	15.0	10.3	13.4	11.0	92.8	93.6
Claude 3.5 Haiku	NONE	61.5	63.0	10.0	12.0	10.0	9.0	2.0	1.5
	NORMAL	15.0	15.0	61.6	63.3	9.0	8.1	1.5	1.3
	DANGEROUS	10.0	12.0	12.0	11.5	71.4	73.1	1.6	1.4
	SYS_OR_SIG	13.5	10.0	16.4	13.2	9.6	9.8	94.9	95.8
LLama 3.3 (70B)	NONE	62.3	60.3	6.0	10.0	10.0	11.0	2.7	2.4
	NORMAL	15.0	14.0	82.4	75.6	9.0	9.0	2.6	2.3
	DANGEROUS	12.0	13.0	7.0	6.3	73.3	71.8	2.6	2.4
	SYS_OR_SIG	10.7	12.7	4.6	8.1	7.7	8.2	92.1	92.9
DeepSeek-r1 (70B)	NONE	70.8	65.4	10.0	9.9	8.0	10.0	3.4	2.4
	NORMAL	15.0	9.0	69.4	73.2	9.0	9.4	2.6	2.1
	DANGEROUS	5.0	12.0	13.0	7.5	73.3	71.2	2.4	2.1
	SYS_OR_SIG	9.2	13.6	7.6	9.4	9.7	9.4	91.6	93.4
Gemma 3 (27B)	NONE	63.3	59.2	11.0	12.0	10.0	10.0	3.0	3.0
	NORMAL	15.0	15.0	71.2	67.0	11.8	12.8	2.0	1.8
	DANGEROUS	12.0	14.0	9.0	10.0	70.6	69.7	2.3	1.9
	SYS_OR_SIG	9.7	11.8	8.8	11.0	7.6	7.5	92.7	93.3
gpt-4o-mini	NONE	58.2	52.6	12.0	10.0	13.6	9.0	4.0	3.0
	NORMAL	14.0	12.0	63.6	61.9	10.0	9.0	3.0	2.5
	DANGEROUS	10.0	10.0	16.6	9.0	68.4	72.7	2.3	1.7
	SYS_OR_SIG	17.8	25.4	7.8	19.1	8.0	9.3	90.7	92.8

3.7 Sonnet as the default model for AC prediction in our subsequent analyses of custom ROMs.

6.4 What is the effect of the similarity score on coverage and accuracy?

Table 6.3: Impact of Cut-off Criteria

Cutoff	None	Normal	Dangerous	Sys	Coverage
0.65	67.8	67.5	73.3	92.3	95.3
0.70	71.6	66.7	57.1	92.8	90.1
0.75	73.2	71.7	79.3	91.0	80.1

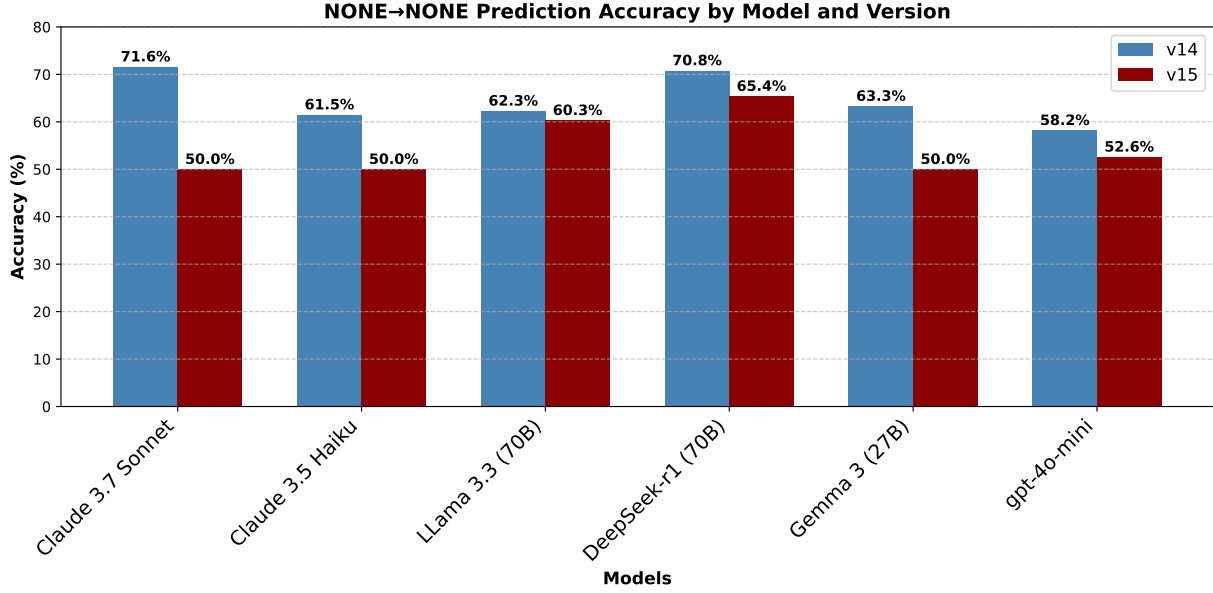


Figure 6.1: None-None prediction accuracy

For this RQ, we evaluate the effectiveness of our tool on Android framework APIs. We use three different similarity score cut-offs (0.65, 0.70, and 0.75) on v14 of AOSP. If the similarity between two sinks is higher than the cut-off score, we treat them as similar and use the top 4 most similar APIs to make AC recommendations. If no similar APIs with matching focal points or sinks are found, our tool cannot provide an AC recommendation.

Table 6.3 shows how different cut-off values affect the accuracy and coverage of our tool. Coverage means how many APIs our tool can analyze, and accuracy means how often the AC recommendations are correct.

We observed that a cut-off of 0.65 gives very high coverage (95.3%), but the accuracy for APIs with no permission (NONE) is lower at 67.8%. When we increase the cut-off to 0.70, the coverage slightly decreases to 90.1%, but the accuracy for NONE improves to 71.6%. Although the accuracy for DANGEROUS APIs drops significantly at this cut-off, overall, the results are balanced, providing good accuracy for NONE, NORMAL, and SYS_OR_SIG categories. At the highest cut-off of 0.75, the coverage decreases significantly to 80.1%, but the accuracy for most categories, especially DANGEROUS (79.3%) and NORMAL (71.7%), improves.

The best balance between accuracy and coverage was at a similarity score of 0.70. This cut-off gave decent coverage (90.1%) and good accuracy across most API categories, making

it a practical choice for use in custom ROMs.

6.5 Can Argus detect AC inconsistencies in custom ROMs?

After selecting Claude Sonnet 3.7 and a similarity score threshold of 0.70 based on the evaluation results from our AOSP analysis, we tested our tool on five different custom ROMs: OnePlus 12, Infinix Smart 8, Vivo Y33s, Samsung A16, and Amazon Tab. For each of these ROMs, we ran the tool on the custom APIs and reused the AOSP APIs and their corresponding AC that had already been analyzed previously.

Table 6.4: Coverage of Custom APIs across ROMs

ROM	Total APIs	Coverage (%)	No Similar APIs
OnePlus 12	737	88.9	82
Infinix Smart 8	117	84.6	18
Vivo Y33s	607	84.7	93
Samsung A16	948	90.2	93
Amazon Tab	196	93.3	13

Table 6.4 summarizes the coverage of our analysis across these ROMs. Coverage refers to the percentage of custom APIs for which our tool was able to find similar APIs with a similarity score above 0.70, allowing for an AC recommendation.

For a subset of APIs, no similar API exceeded our similarity threshold of 0.70. Consequently, for these APIs, our tool could not recommend an appropriate AC.

To ensure the accuracy and validate the practical utility of our tool, we manually analyzed 10% of the detected inconsistencies from each ROM. The selected inconsistencies were reviewed to categorize them as (1) True Positive: The API does not enforce AC on a sensitive operation and is therefore a real security issue. (2) False Positive: The flagged API does not pose a security threat, and therefore does not require AC, or (3) Unresolved Positives: We could not make a definite decision. In these cases, the API might belong to a proprietary feature or vendor-specific logic, and it would require deeper knowledge of the internal codebase to confirm.

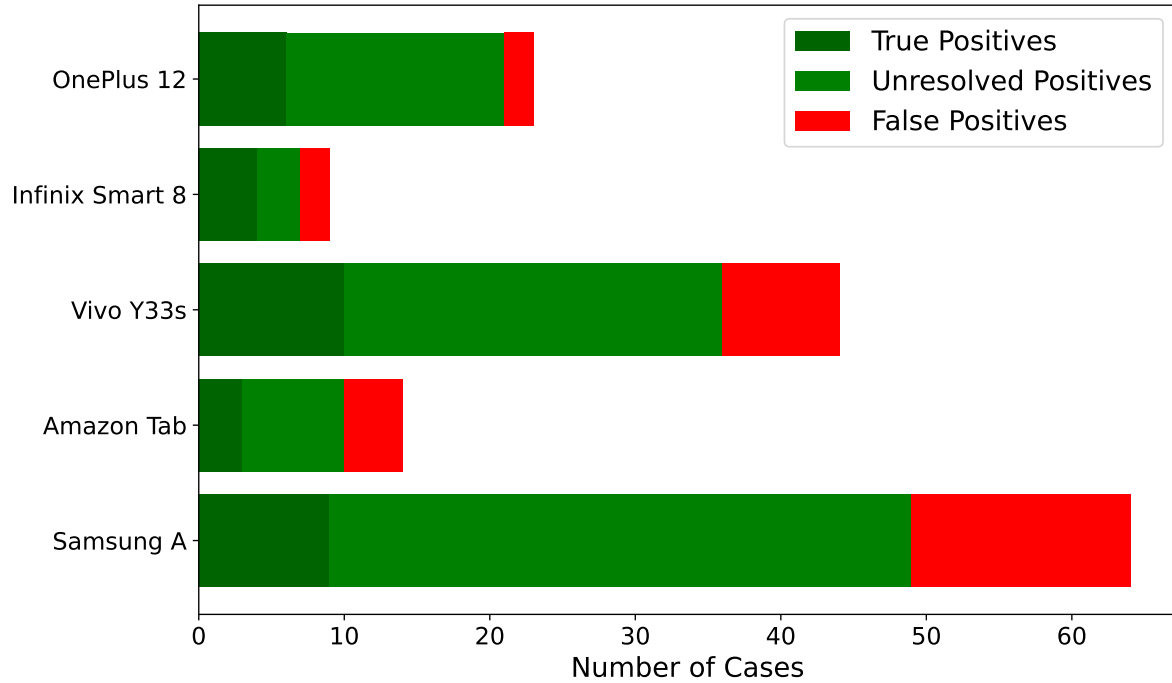


Figure 6.2: Categorization of Positives Reported by ARGUS

As shown in figure 6.2, ARGUS reported an average True Positive (TP) rate of 25.75%, meaning that approximately one-fourth of the APIs flagged by our tool were indeed genuine security issues lacking proper access control enforcement. Additionally, the average rate of Unresolved Positives stood at 54.03%, indicating that more than half of the flagged APIs required deeper inspection or vendor-specific knowledge for conclusive evaluation. The False Positive (FP) rate averaged at 20.22%, showing that a relatively small proportion of APIs were incorrectly flagged by our system as potential security concerns.

To demonstrate the practical security impact of confirmed true positives, we developed end-to-end proof-of-concept (PoC) exploits. These exploits highlight significant security risks and are detailed in Table 6.5.

Among the vulnerabilities identified and demonstrated were: A complete service in Vivo Y33s that allows access to sensitive device lock and SIM state information, APIs in OnePlus 12 that indirectly revealed all installed package names through unprotected refresh rate settings and permitted unauthorized modifications of screen brightness and audio configuration, APIs in Infinix Smart 8 that enabled unauthorized apps to alter critical

Table 6.5: Summary of Discovered Vulnerabilities

ROM	Service API	Security Implication
Vivo Y33s	ActivityManagerService.setScanValue	Allows unauthorized apps to modify internal camera scan parameters
Vivo Y33s	ActivityManagerService.notifyWechatScanCrash	Allows unauthorized apps to spoof WeChat crash reports
Vivo Y33s	VivoPrivVoclController.getVivoLockState isSimLockSupport getOperatorCustomize isPasswordPreset getUnlockMaxCount isDeviceLockSupport isOperatorMatchedCard isInLockMode isRadioDowning	Expose privileged device lock and SIM status information to unauthorized apps
Oneplus 12	DisplayManagerService.setSpecBrightness	Modify screen brightness parameters without restriction
Oneplus 12	OplusDisplayModeService. removeAllCustomizeRefreshRate, removeCustomizeRefreshRate	Remove custom refresh rate settings for all or specific apps
Oneplus 12	OplusDisplayModeService. getAppOverrideRefreshRateList	Reveals all installed packages with associated refresh rate settings
Oneplus 12	AudioService.cacheBinauralRecordParameters, cacheParameters	Allows unprivileged apps to modify internal audio configuration cache
Oneplus 12	OplusDisplayModeService.setAppOverrideRefreshRate	Enables unauthorized apps to override refresh rate of apps
Amazon Tablet	ConfigUpdateInstallReceiver.onReceive	Allows unauthorized apps to trigger privileged configuration updates
Infinix Smart 8	TranssionWifiServiceImpl.getBlockedList	Exposes user-defined blocked MAC address list
Infinix Smart 8	TranssionWifiServiceImpl.setBlockClient	Allows unauthorized apps to overwrite the blocked client list
Infinix Smart 8	TranssionWifiServiceImpl.setMaxClient	Allows unprivileged apps to change Wi-Fi hotspot client limit
Infinix Smart 8	AppWidgetServiceImpl.getBoundAppWidgets	Reveals apps with active widgets on the homescreen
Samsung A16	KnoxCustomManagerService.getFavoriteApp	Returns the user's favorite app, exposing personal preferences
Samsung A16	KnoxCustomManagerService.getAutoCallNumberList	Get list of auto call numbers setup by a user
Samsung A16	AccessibilityManagerService semDisableWindowMagnification, setMagnificationDisactivate	Disables user-enabled magnification settings, potentially interfering with accessibility features

Wi-Fi hotspot settings, including changing client blocklists and adjusting the maximum number of connected clients.

These findings underscore the effectiveness of ARGUS in identifying critical security vulnerabilities and emphasize the importance of accurate AC implementations in custom ROMs. Several of these reported vulnerabilities have already been acknowledged by the respective vendors, affirming their relevance and impact.

6.6 How does Argus compare to previous approaches?

6.6.1 Convergence-Based Approach

Several previous works used convergence-based analysis to detect inconsistent AC enforcement in the Android framework. We compare ARGUS to the state-of-the-art convergence-based tool, ACMiner [9]. ACMiner evaluates its approach on an AOSP Nexus 5X running v7.1.1. The paper manually validates all 246 reported cases to determine whether they are true positives, ultimately identifying 28 confirmed inconsistencies.

We select ACMiner as the baseline because this Nexus 5X dataset is the only publicly available benchmark with detailed per-API results and manual validation, enabling a direct case-by-case comparison. Since ACMiner does not run on our collected ROMs, we run ARGUS on the same Nexus 5X dataset and report the results below.

Rediscoveries. Out of the 28 inconsistencies reported by ACMiner, ARGUS successfully rediscovers 24. Some examples include `getInstalledApplications`, `hasUserRestriction` and `installExistingPackageAsUser`.

For `getInstalledApplications`, ARGUS identifies the focal functionality and retrieves four highly similar sinks in `getInstalledPackages` within the same ROM, all protected by a system-level permission. Since both methods access the same sensitive package database and return detailed information on all installed applications, ARGUS correctly infers that `getInstalledApplications` should also be protected by a system level permission.

Similarly, for `hasUserRestriction`, ARGUS identifies similar sinks in `hasBaseUserRestriction` and `setUserRestrictions`, both protected by system AC. As the method accesses system policy data on user restrictions, information that is integral to device security, ARGUS correctly infers that it warrants the same system protection.

Missing Inconsistencies. Two ACMiner-reported APIs—`getMonitoringTypes` and `getStatusOfMonitoringType`—could not be analyzed by ARGUS due to a DEX-file parsing

issue with WALA, the static-analyser that ARGUS uses. For `convertFromTranslucent`, no top-ranked similar APIs were retrieved above the similarity threshold, and hence ARGUS could not make an AC recommendation. The last API, `logEvents` is already protected by a system level check, whereas ACMiner recommends a *different* system-level permission. ARGUS evaluates consistency at the granularity of the four levels and flags a case only when the inferred level is *stronger* than the enforced one (e.g., normal vs. system).

New Inconsistencies. We select 10% of the reported inconsistencies and manually analyze them, classifying each as True Positive or False Positive. The total number of new inconsistencies identified are 18. These are the cases that are identified to be protected in the latest version of Android (AOSP v15). Some examples of new vulnerabilities identified include `removeUser`, `getUserRestrictions`, `getPhoneId`, and `getActiveSubIdList`. These are identified by ARGUS since the convergence points identified by ACMiner are either auxiliary (e.g., logging, cache/state updates) or fail to converge on a single sink. In contrast, ARGUS successfully finds the true focal sinks and aligns them with protected AOSP counterparts: `removeUser` maps to user-deletion routines; `getUserRestrictions` returns the effective restrictions bundle; and `getPhoneId` / `getActiveSubIdList` expose telephony/subscription identifiers. Where the target APIs lack comparable enforcement as they enforce a weaker level than their similar APIs, ARGUS infers an inconsistency via semantic sink matching rather than relying on auxiliary or irrelevant convergence points.

TP / FP. As this is an AOSP ROM, for each reported vulnerability, we check whether the API is protected in the latest version of Android, AOSP v15. If the API is protected in v15, we classify it as TP; otherwise, it is marked FP. Using this procedure, ARGUS achieves an FP rate of 49.6%, whereas ACMiner exhibits a much higher FP rate of 93.4%. This difference stems from the convergence-based assumption that every convergence point is the intended target of AC, which often includes auxiliary or non-sensitive code paths and thereby inflates false positives. In contrast, ARGUS combines focal functionality extraction with semantic similarity retrieval, reducing the likelihood of incorrectly flagging benign operations.

6.6.2 Probabilistic Approach

We further compare ARGUS to the state-of-the-art probabilistic tool, Poirot [5]. Poirot models the resource–AC association as *uncertain* and combines path-sensitive facts (control dependencies, reachability, trigger conditions, naming hints, mutual exclusion) with a probabilistic inference engine to propagate protections, thereby reducing false positives from non-sensitive convergence points.

Poirot evaluates its probabilistic approach on several devices, including the Amazon Fire HD, which is an OEM-customized Android ROM that introduces various custom APIs. Since Poirot’s implementation and per-API outputs are not publicly available, we run ARGUS on the same ROM and base our comparison on the specific cases reported in the paper.

Rediscoveries. ARGUS rediscovers all 5 vulnerabilities previously reported by Poirot. These include `setInputFilter`, `migrate`, `getMigrateData`, `setAmazonFlags` and `removeAmazonFlags`.

For `getMigrateData`, ARGUS identifies the focal functionality that returns system storage migration status and retrieves the following AOSP APIs: `isDreaming` and `getDataBlock Size`, both guarded by system level permission. Given the method’s sensitivity, ARGUS correctly infers that `getMigrateDataInfo` should also be protected at the system level AC.

Similarly, For `setAmazonFlags`, ARGUS matches it to the following custom APIs: `setAmazonMetadataForUser` and `removeAmazonMetadataForUser`, both of which are protected with system protection and that modify Amazon application metadata for users. Since the method changes system-level data structures and writes to files, ARGUS similarly concludes that it warrants system protection.

New Inconsistencies. Beyond these rediscoveries, ARGUS flags additional inconsistencies. Some of these include `magnificationCanvasAddLine`, `magnificationCanvasAddRect` and `magnificationCanvasClear`. ARGUS finds the *focal sinks* that change the magnification, filters out auxiliary logging/validation, and then retrieves semantically aligned AOSP accessibility/magnification setters (e.g., enabling/disabling magnification, resetting magnification state) that are guarded at system level permission. On the other hand, the probabilistic tools suffer when observable hints (e.g., trigger conditions or control-dependency) are sparse, and therefor, struggle to recommend high-confidence AC for these APIs.

TP / FP. Sampling 10% of the remaining ARGUS-reported cases yields an estimated false-positive rate of 57.4% (i.e., lower than Poirot’s reported 76.4%).

Coverage. ARGUS makes an AC recommendation for 76.8% of candidate APIs on Amazon Fire HD. Poirot does not report coverage for this device; it reports approximately 60% coverage on AOSP, suggesting that ARGUS provides broader coverage in practice.¹ We attribute ARGUS’s improvements to its combination of focal-sink extraction and semantic retrieval, which focuses LLM reasoning on security-relevant operations rather than assuming all convergence candidates are targets of AC.

¹Coverage figures are not strictly comparable across devices; we include the AOSP number as the closest reported reference point.

Sink identification and coverage. Poirot’s probabilistic target inference reduces false positives by scoring candidate resources with manually designed features and implication rules, but it can miss semantically implied targets when these signals are sparse or ambiguous, and its coverage is reported to be limited (e.g., $\sim 60\%$ on AOSP) [5]. In contrast, ARGUS leverages LLM-based code analysis (guided by Android-specific positive/negative rules) to filter auxiliary operations and elevate focal functionality, and then uses code embeddings to retrieve semantically aligned examples. This combination surfaces relevant sinks more reliably and provides richer context for AC inference. Empirically, this correlates with higher accuracy and coverage on AOSP (84.5–84.9% accuracy with 90.1–90.7% coverage; §6.1.2–§6.1.3) and broader coverage on Amazon Fire HD (76.8%). While a strict comparison is constrained by Poirot not being open source and by device/setup differences, these results suggest that LLM-guided sink identification plus semantic retrieval can identify more relevant cases at higher coverage than probabilistic target inference alone.

Chapter 7

Case Studies

In this section, we discuss two detailed case studies that illustrate the capabilities of ARGUS in detecting AC vulnerabilities, highlighting scenarios that existing inconsistency detection tools typically struggle to identify accurately.

7.1 Unauthorized Refresh Rate Modification in OnePlus 12

The first case study pertains to the vulnerability identified in the OnePlus 12 involving the custom API `setAppOverrideRefreshRate`, which permits applications to set their refresh rate without explicit user consent. ARGUS analyzed the identified sinks within this API and subsequently retrieved two highly similar APIs: (1) the `setRefreshRateSwitchingType` (I)V from the `DisplayManagerService` class in AOSP, enforcing a `SYS_OR_SIG` permission; and (2) `setExternalRefreshRateStatus` (I)V from the custom `OPlusRefreshRateService` class in the OnePlus ROM, also requiring `SYS_OR_SIG`. Leveraging the retrieved similar APIs and their respective AC levels, the LLM accurately recommended that `setAppOverrideRefreshRate` should be protected by a `SYS_OR_SIG` permission.

ARGUS was also effective in flagging additional similar vulnerabilities within the same service. For example, the API `getAppOverrideRefreshRateList`, which returns a list of all installed packages with associated refresh rate configurations, was similarly identified as unprotected. All identified vulnerabilities in this case have been responsibly reported to OnePlus for acknowledgment.

7.2 Exposure of Sensitive SIM and Device Lock Information in Vivo

The second case study focuses on the `VivoPrivVoclController` service found within the Vivo ROM, responsible for handling various SIM and device lock functionalities. ARGUS flagged the entire set of APIs within this service as vulnerable due to unauthorized exposure of sensitive SIM lock and device status information. Specifically, APIs such as `isSimLockSupport()`, `getOperatorCustomize()`, `isPasswordPreset()`, `getUnlockMaxCount()`, `isDeviceLockSupport()`, `isOperatorMatchedCard(I)`, `isInLockMode(III)`, and `isRadioDowning(I)` were identified as lacking proper AC, thus potentially allowing unauthorized applications to access sensitive system information related to SIM and device lock.

Upon analysis, ARGUS successfully retrieved several semantically similar APIs both from the Vivo custom ROM and AOSP. Relevant examples include the custom APIs `getCustomOperatorNameForEsim`, `getSubSIMAllowList`, and `getIsUserSubSIMSwitchOn()`, each requiring a `NORMAL` permission. Additionally, an AOSP API, `getState()` from `BluetoothManagerService` class, enforcing `SYS_OR_SIG`, was identified as semantically similar, retrieving the state.

It is important to note here that conventional convergence based and probabilistic-based detection tools would provide incorrect recommendations in this scenario, as all APIs within the `VivoPrivVoclController` service are similarly unprotected and utilize related data fields and native methods. These tools would mistakenly infer uniform permissions due to shared fields and resources. In contrast, ARGUS utilizes semantic similarity retrieval techniques that accurately discern the nuanced differences in API functionalities and data flows, enabling it to recommend the correct `NORMAL` permission, thereby effectively mitigating potential security threats.

Chapter 8

Related Works

8.1 Vulnerability Detection Using LLMs

Recent research has demonstrated that LLMs are highly effective in identifying vulnerabilities in codebases. Fang et al.[6] showed that LLMs can successfully reason about complex control and data flows to uncover vulnerabilities even when the code is incomplete or slightly obfuscated. Lin et al.[15] developed an approach leveraging LLMs to detect security flaws by understanding API semantics deeply, significantly outperforming traditional pattern-matching tools. Wang et al. [21] introduced LLM-based data-flow analysis to pinpoint subtle bugs across extensive codebases, demonstrating the capability of these models to understand intricate interactions between code components.

Liu et al.[16] further enhanced vulnerability detection by using LLM-generated code patches, which can both detect vulnerabilities and propose effective remediation strategies. Li et al.[14] integrated LLMs into static analysis pipelines to reduce false positives substantially, highlighting the model’s ability to validate actual vulnerabilities accurately. Additionally, Zheng et al. [22] proposed autonomous LLM-driven auditing tools that utilize agent-based memory and sophisticated data-flow tracking techniques to efficiently locate real-world vulnerabilities within large repositories, thereby significantly improving the efficiency of vulnerability detection.

8.2 Inconsistent Access Control Detection in Android

Early studies like Stowaway [7] focused on dynamic testing to determine discrepancies between requested permissions and actual permissions needed by Android applications. Subsequently, Pscout [2] utilized static reachability analysis to map permissions required by framework APIs. Aexplorer by Backes et al. [3] enhanced static analysis techniques by constructing detailed runtime models to precisely identify permissions.

To improve the accuracy further, Arcade [1] performed path-sensitive static analysis using innovative graph extraction techniques. Dynamo [4] applied dynamic fuzzing to generate accurate permission requirements. These advancements set the stage for tools like Kratos [18], Acedroid [1], and ACMiner [9], which employ convergence-based analysis. These tools identify paths leading to common points (convergence points) and flag inconsistencies if they have different AC levels. However, these tools tend to produce high false positives because auxiliary instructions such as logging or debugging can falsely indicate sensitivity.

To mitigate these false positives, Poirot [5] introduced probabilistic inference. Unlike convergence-based tools, Poirot treats the AC-resource relationship as uncertain. It aggregates multiple hints from data-flow, reachability, and semantic clues to probabilistically infer accurate AC recommendations. Despite this innovation, Poirot faces limitations due to manually defined inference rules and preset probability values, making it less sensitive to nuanced semantics and causing relatively low coverage (around 60%).

Additionally, Gorski [10] used semi-automated methods to uncover multi-user vulnerabilities, highlighting broader inconsistency issues in AC enforcement. FReD [11] expanded on previous techniques by merging Linux-layer file access control with Android-layer permissions, detecting further multi-user vulnerabilities. IAceFinder [23] analyzed inconsistencies across Java and native layers, enhancing cross-layer detection capabilities. Bluebird [19] combined app-side logic and static semantics via probabilistic inference to audit Android API protections comprehensively.

Given the limitations identified in current AC recommendation systems, including convergence based approaches, probabilistic tools like Poirot, and initial attempts using LLMs, we propose a hybrid approach that combines static analysis, code embeddings, and LLMs to accurately recommend AC. To address the limitations of convergence based tools that often produce high false positives by assuming every convergence point as sensitive, first, we utilize static analysis to prepare a summarized program execution from the API bytecode. Next, instead of relying on the simplistic convergence assumption, our tool leverages the code analysis abilities of LLMs, providing them with carefully structured Android-specific rules. This allows LLMs to differentiate between actual sensitive operations from benign or

auxiliary code. Moreover, probabilistic tools like Poirot suffer from limited coverage and accuracy due to manually defined inference rules and fixed probability scores, which can overlook nuanced Android semantics. Our approach overcomes these issues by employing code embeddings, generated using models such as CodeT5+, to effectively capture both semantic and syntactic information of API execution paths. By performing similarity searches on these embeddings, our retrieval system can accurately identify APIs with contextually similar behaviors and known correct AC. This collective analysis significantly improves the contextual understanding necessary for accurate AC recommendations.

Chapter 9

Limitations and Future Work

9.1 Diverse Access Control Checks

In this research, we primarily focused on encoding the most common static AC patterns typically present in AOSP and frequently observed in custom ROMs. However, a significant limitation arises from the highly diverse nature of AC implementations across customized ROMs. Vendors often introduce unique security mechanisms and custom AC enforcement methods that significantly deviate from standard patterns identified in AOSP.

These vendor-specific AC implementations may use complex, proprietary checks. For instance, some vendors employ unique User Identifiers (UIDs) for privileged processes, for instance by Samsung’s Knox security, which uses custom UIDs to enforce specific AC. Such vendor specific customizations present a notable challenge for our static analysis approach, as our tool may not fully capture or recognize these unique patterns due to their deviation from standard AOSP conventions.

Moreover, certain framework services in Android are protected using Security-Enhanced Linux (SELinux) policies. SELinux provides a mandatory AC system that defines policies to enforce restrictions on processes, services, and resources. These policies effectively prevent third-party applications from invoking certain sensitive APIs within protected services, significantly reducing potential exploitation vectors. Our current analysis pipeline does not fully incorporate SELinux policy analysis, limiting the scope and comprehensiveness of our vulnerability detection. Thus, an API flagged as vulnerable might be practically secure due to an implemented SELinux protection.

9.2 Selecting the Longest Path for Focal Functionality Identification

In our current implementation, the analysis pipeline involves selecting the longest execution paths at each depth level (depths 1, 2, and 3) to be analyzed further by the tool. This approach is based on the assumption that the longest paths generally encapsulate the most comprehensive details about method invocations, data-flow, and potential security-sensitive operations (hence sinks). Consequently, these paths are assumed to offer a richer context for the LLM to identify and analyze the focal functionalities critical for accurate AC recommendations.

However, this heuristic approach inherently contains some limitations. Primarily, it assumes that the length of an execution path directly correlates with its relevance or significance regarding sensitive operations. It might be possible that shorter execution paths may sometimes include more critical functionalities than their longer counterparts, especially if the longer paths predominantly consist of auxiliary or benign operations such as debugging statements, state update or irrelevant method invocations. This scenario would potentially result in the omission of crucial security-sensitive operations during analysis, thereby affecting the accuracy of AC recommendations.

To mitigate this issue, future work could integrate LLM-driven methods to intelligently select the most contextually relevant execution paths rather than merely the longest ones. Specifically, at each depth, an LLM could be prompted to evaluate multiple candidate paths and identify the path that most likely contains the critical focal functionality based on semantic and data-flow analyses. This selected path could then be expanded upon in subsequent depths, iteratively refining the analysis and ensuring more comprehensive coverage of genuinely sensitive operations. Such an approach would enhance the precision of our vulnerability detection pipeline, effectively reducing false positives and improving the reliability of the recommended AC. We leave this as future work.

Chapter 10

Conclusion

This thesis presents a comprehensive approach to addressing AC inconsistencies within Android framework APIs, specifically focusing on vulnerabilities introduced through vendor customizations in custom ROMs. By leveraging a hybrid analysis pipeline that integrates static code analysis, code embedding, and context-aware LLMs, our proposed system, ARGUS, effectively enhances the precision of AC recommendations.

Our evaluation across five custom ROMs; OnePlus 12, Vivo Y33s, Samsung A16, Amazon Tab, and Infinix Smart 8, demonstrates ARGUS’s effectiveness by identifying 32 unique AC vulnerabilities. Among these, we successfully developed proof-of-concept exploits for 28 cases, clearly showcasing the real-world security implications of these inconsistencies. Several of these vulnerabilities have already been acknowledged by respective vendors, validating the practical impact and necessity of such robust detection mechanisms.

The results present the potential of integrating LLMs with traditional static analysis methods, highlighting the significant improvements in accuracy (84.5%–84.9%) and coverage (90.1%–90.7%) compared to existing approaches. Moving forward, this research contributes valuable insights and tools to the Android security community, paving the way for future advancements in securing Android’s increasingly complex and customized ecosystem.

References

- [1] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. Acedroid: Normalizing diverse android access control checks for inconsistency detection. In *NDSS*, 2018.
- [2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.
- [3] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: {Re-Visiting} android permission specification analysis. In *25th USENIX security symposium (USENIX security 16)*, pages 1101–1118, 2016.
- [4] Abdallah Dawoud and Sven Bugiel. Bringing balance to the force: Dynamic analysis of the android application framework. *Bringing balance to the force: dynamic analysis of the android application framework*, 2021.
- [5] Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. Poirot: Probabilistically recommending protections for the android framework. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 937–950, 2022.
- [6] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. Large language models for code analysis: Do {LLMs} really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846, 2024.
- [7] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

- [8] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The L^AT_EX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.
- [9] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. Acminer: Extraction and analysis of authorization checks in android’s middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 25–36, 2019.
- [10] Sigmund Albert Gorski III. *Semi-automated Evaluation of Access Control Enforcement in the Android Platform*. North Carolina State University, 2020.
- [11] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. {FReD}: Identifying file {Re-Delegation} in android system services. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1525–1542, 2022.
- [12] Donald Knuth. *The T_EXbook*. Addison-Wesley, Reading, Massachusetts, 1986.
- [13] Leslie Lamport. *L^AT_EX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.
- [14] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):474–499, 2024.
- [15] Jie Lin and David Mohaisen. From large to mammoth: A comparative evaluation of large language models in vulnerability detection. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*, 2025.
- [16] Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating api parameter security rules with llm for api misuse detection. *arXiv preprint arXiv:2409.09288*, 2024.
- [17] ollama. ollama, Accessed on: May 28, 2025. <https://github.com/ollama/ollama>.
- [18] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.
- [19] Parjanya Vyas, Asim Waheed, Yousra Aafer, and N Asokan. Auditing framework {APIs} via inferred app-side security specifications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6061–6077, 2023.
- [20] WALA. Wala, Accessed on: November 27, 2024. <https://github.com/wala/WALA>.

- [21] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. Llmdfa: analyzing dataflow in code with large language models. *Advances in Neural Information Processing Systems*, 37:131545–131574, 2024.
- [22] Mingwei Zheng, Danning Xie, and Xiangyu Zhang. Large language models for validating network protocol parsers. *arXiv preprint arXiv:2504.13515*, 2025.
- [23] Hao Zhou, Haoyu Wang, Xiapu Luo, Ting Chen, Yajin Zhou, and Ting Wang. Uncovering cross-context inconsistent access control enforcement in android. In *The 2022 Network and Distributed System Security Symposium (NDSS'22)*, 2022.