RULEDROID: LLM-Augmented Synthesis of Static Security Detection Rules for Android Apps

Zhentao Xie, Mingyang Chen, Yaqi Gao, Shishuai Yang, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang

***Abstract*—Android's vast ecosystem and expansive API surfaces pose a serious challenge to static application security testing (SAST) tools. Mainstream tools such as MobSF, APKHunt, and AUSERA mainly rely on manually crafted rules. Crafting these rules demands considerable effort, yet they still cannot cover every security issue. When Android introduces new APIs, changes its permission model, or revises other security policies, the rules soon fall behind. Without constant maintenance, false positives grow, and true vulnerabilities slip through. Recently released LLM-based detectors are easy to use and potentially support a wide range of vulnerability types, but their findings often lack clear explanations and suffer from high false-positive rates.**

In this paper, we present RULEDROID, a new framework that leverages LLMs to automatically generate Semgrep-compatible static detection rules from up-to-date official Android security documentation. RULEDROID tackles the limits of pure LLM detection by combining (i) Retrieval-Augmented Generation (RAG), which grounds model outputs in trusted documents, and (ii) a modular workflow that decomposes rule synthesis into well-defined stages. The resulting rules are then applied with proven static-analysis techniques, ensuring consistent and explainable results. We evaluated RULEDROID on three public benchmark datasets. Based on its large and precise rule set, RULEDROID achieved higher coverage and accuracy than traditional SAST tools, and sharply reduced false positives compared with direct LLM scanning. When applied to real-world apps, RULEDROID discovered multiple new vulnerabilities, resulting in 57 CVE IDs being assigned. These results show that RULEDROID combines the broad vulnerability coverage of LLMs with the precision of static analysis, delivering a fully automated docs-to-rules solution for Android security testing.

***Index Terms*—Android app security, vulnerability detection, automated rule generation, LLM4SE**

This work was supported in part by the Natural Science Foundation of Shandong Province under Grants ZR2025MS991 and in part by the Open Research Fund of the State Key Laboratory of Internet Architecture under Grant HLW2025MS29. *(Corresponding author: Wenrui Diao.)*

Zhentao Xie is with the School of Cyber Science and Technology, Shandong University, Qingdao 266237, China, and also with the State Key Laboratory of Internet Architecture, Tsinghua University, Beijing 100084, China (e-mail: fxizenta@mail.sdu.edu.cn).

Mingyang Chen and Yaqi Gao are with the School of Cyber Science and Technology, Shandong University, Qingdao 266237, China (e-mail: ocean.cmy@mail.sdu.edu.cn; gyq2022@mail.sdu.edu.cn).

Shishuai Yang is with the School of Computer Science, Zhengzhou University of Aeronautics, Zhengzhou 450015, China (e-mail: shishuai@zua.edu.cn).

Wenrui Diao is with the School of Cyber Science and Technology, Shandong University, Qingdao 266237, China, and also with the State Key Laboratory of Internet Architecture, Tsinghua University, Beijing 100084, China (e-mail: diaowenrui@link.cuhk.edu.hk).

Xiangyu Liu is with the Security Department, Alibaba Group, Hangzhou 310052, China (e-mail: eason.lxy@alibaba-inc.com).

Kehuan Zhang is with the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong, China (e-mail: khzhang@ie.cuhk.edu.hk).

I. INTRODUCTION

Smartphones play a central role in daily life, and Android apps offer significant convenience in areas such as shopping, social networking, and payments. As these apps have grown more complex, they now store large volumes of sensitive user data, exposing users to serious security risks, including unauthorized access and data breaches. Therefore, ensuring the security and reliability of Android apps is a real-world necessity.

Since Android apps remain prime targets for security vulnerabilities, extensive research and tool development in static analysis have been motivated by this threat. Several Static Application Security Testing (SAST) tools have been developed in both academia and industry to detect app flaws without executing code. Tools such as MobSF [1], APKHunt [2], and AUSERA [3] typically follow a workflow in which the app is decompiled, the code or manifest is analyzed using predefined security rules, and any matches are reported. While static scanners are efficient for large-scale analysis, ***a fundamental limitation lies in their reliance on manually crafted and fixed rule sets***. On the one hand, these tools often focus only on high-profile vulnerabilities while overlooking others. On the other hand, the Android platform is continuously evolving, with each release introducing new APIs and permissions, deprecating others, updating default configurations, and exposing new classes of vulnerabilities. As a result, maintaining accurate and comprehensive rule sets requires continuous human effort, which is both time-consuming and difficult to sustain.

In recent years, the rise of Large Language Models (LLMs) has opened new possibilities for code analysis [4]. Advanced LLMs, such as GPT-5, have demonstrated strong code understanding and the ability to identify certain vulnerabilities through learned patterns, spurring research into LLM-driven vulnerability detection [5], [6], [7], [8], [9], [10]. An LLM-based approach might generalize beyond a fixed rule set by "reasoning" about code semantics. However, while LLMs offer rich pattern-recognition and reasoning capabilities, their direct application to security detection is limited by inherent issues [11]. LLMs generate outputs probabilistically, sometimes producing inconsistent results or "hallucinations" [12], and they struggle with processing large codebases due to token length limits and the attention mechanism [13]. Consequently, an approach solely based on LLM outputs does not guarantee comprehensive or reliable detection, especially in mission-critical security contexts.

Our Work. To bridge this gap, we propose RULEDROID, an automated vulnerability detection framework that combines the

strengths of LLMs and traditional static analysis. RULEDROID automatically generates detection rules by leveraging LLMs to parse official Android security documentation [14] and extract relevant safety guidelines. The generated rules are then converted into a Semgrep-compatible format (a well-established static analysis framework), ensuring that the final vulnerability detection is performed using deterministic, interpretable rules. This design choice not only avoids the instability of LLM outputs but also allows the detection system to be easily updated in response to changes in Android security requirements. To further improve performance, we employ the overall design of a workflow [15] to break the complex rule generation task into smaller steps, and we use Retrieval-Augmented Generation (RAG) knowledge bases [16] to provide targeted support to each LLM stage. This approach substantially reduces hallucinations and improves the quality of LLM-generated rules [17].

RULEDROID was evaluated on three well-established benchmarks, including Ghera, MASTG&PIVAA, and the DroidCVE++ dataset, which aggregates real-world vulnerabilities from historical CVEs. Experimental results indicate that RULEDROID outperforms existing SAST tools (including SPECK, MobSF, Marvin, AndroBugs, APKHunt, AUSERA, and QARK) and LLM-driven detectors (based on four state-of-the-art LLMs). For example, on the Ghera benchmark, the best-performing traditional scanner (APKHunt) detects only 28 out of 59 known issues, whereas RULEDROID detects 40 issues. On the DroidCVE++ dataset, RULEDROID achieves a recall of 90.29%, dramatically higher than GPT-5's 65.11%. RULEDROID also leads in precision and F1-score across benchmarks, indicating a well-balanced detection capability. Furthermore, in the scan of real-world Android apps, we confirmed 417 true cases from 500 randomly selected `ERROR`-level issues. We then responsibly reported 62 exploitable cases, and the assessment process is ongoing. To date, 57 CVE IDs (e.g., `CVE-2025-7889`, `CVE-2025-7890`) have been assigned, demonstrating the practical effectiveness of our approach. The complete list of CVE IDs can be found in Appendix A.

Contributions. The main contributions of this paper are:

- ***Automated Rule Generation.*** RULEDROID is the first framework to dynamically generate Semgrep-compatible static detection rules by mining official Android security documentation. This automation removes the need for time-consuming manual rule writing and ongoing maintenance.
- ***Comprehensive Evaluation.*** Evaluations on three public benchmarks show that RULEDROID outperforms both traditional static analysis tools and LLM-based methods. In particular, RULEDROID achieves much higher recall, making it well-suited for real-world Android app security analysis. Also, multiple real-world vulnerabilities were discovered.
- ***Enhanced Benchmark.*** We introduce DroidCVE++, an extended benchmark that expands existing CVE-based datasets by covering more vulnerability types and providing precise, easily verifiable ground truth (e.g., vulnerability location details). This benchmark offers a more realistic testbed for future static-analysis research.

Ethical Considerations and Responsible Disclosure. All experiments were performed in a controlled local environment. All verified vulnerabilities have been responsibly reported to the affected vendors or CVE authorities. So far, 57 CVE IDs have been assigned, with more in progress.

Open Science. The prototype of RULEDROID is available at https://doi.org/10.5281/zenodo.17501815. We also release the DroidCVE++ benchmark to promote reproducibility and support further research in the community.

II. BACKGROUND AND MOTIVATION

This section provides the necessary background on Static Application Security Testing (SAST) and Large Language Models (LLMs), along with the challenges they face in detecting vulnerabilities in Android apps. We *emphasize that* this paper focuses on general vulnerability detection in Android apps. It does not cover malware detection, which mainly aims to distinguish between benign and malicious apps, and differs significantly from our goal.

A. Static Application Security Testing

SAST uses static code analysis to examine code structure, data flow, and logic, thereby identifying potential security vulnerabilities without requiring the app to be run. Both academia and industry have developed many static analysis tools for different use cases, including a wide range of SAST tools designed for Android apps. Tools such as MobSF [1], APKHunt [2], and AUSERA [3] typically follow a common workflow. They first decompile the target APK, then analyze the decompiled code using predefined security rules, and finally generate a report based on the results. SAST tools are well-suited for large-scale scanning and can efficiently detect potential security risks in apps.

Current Issues. However, such tools typically ***rely on fixed, static detection rules manually crafted by developers***. Constructing such rules requires tool developers to extract and generalize patterns based on their domain expertise and available technical documentation. However, due to the high cost of manual rule development, coverage is often limited. Many tools focus on high-profile vulnerabilities while overlooking less prominent yet still important issues. For example, SPECK [18] detects only 31 vulnerability types derived from official Android security guidelines.

Also, the Android ecosystem is highly dynamic. Each year, new versions of the Android OS introduce changes to APIs, permission models, and default configurations. At the same time, new types of vulnerabilities continue to emerge, prompting frequent revisions to security requirements and best practices. As a result, developers of SAST tools must invest substantial effort over long periods to track platform changes and keep detection rules up to date. Outdated rules that fail to reflect the latest security standards can lead to both false positives and false negatives. For instance, Marvin [19], which was last updated in 2018, lacks rules for detecting the "StrandHogg Attack," a vulnerability disclosed in 2019 [20]. In contrast, MobSF [1] has continued to receive updates and includes detection for this issue. However, its rules were

not revised after the vulnerability was officially patched in September 2020 [20]. Since Android 11, this vulnerability has been eliminated. If a tool does not account for the app's `minSdkVersion`, it may still flag this vulnerability in apps targeting newer Android versions, resulting in false positives.

B. Large Language Models for Security

LLM is a type of AI model designed to process and generate natural language. It is built on deep learning techniques and trained on large-scale text corpora. Most LLMs adopt the Transformer architecture (such as GPT and BERT) and learn language patterns, grammar, and factual knowledge through extensive training. Recent state-of-the-art models, including GPT-5, o4-mini, DeepSeek-V3.2, and DeepSeek-R1, demonstrate impressive capabilities in understanding and reasoning. With the rapid advancement of LLMs, a growing number of studies are investigating their applications in software security, particularly for detecting vulnerabilities.

In the typical workflow for an LLM-based security detection task, the system first clarifies the task for the LLM with a predefined prompt. Subsequently, the code snippet to be analyzed, along with its relevant context information, is fed into the model. The LLM then generates vulnerability detection results based on the provided input.

Current Issues. Although directly employing an LLM to detect Android code vulnerabilities is straightforward and convenient, several issues persist in practice. First, the most evident problem is that LLM-based detection methods generally exhibit a ***high false-positive rate***, as demonstrated by Ullah et al. [11] and Section V-E. This is attributed to the combined effect of LLMs' statistical properties and other characteristics. Second, ***the capability of LLMs to process long code segments*** is limited. When target program files are large, the analysis may be incomplete. LLMs may even fail to perform any analysis if the file length exceeds the input constraints. Moreover, ***the output of LLMs is non-deterministic***, meaning results may vary even with identical inputs. LLMs also suffer from the "hallucination" problem [12], wherein they may produce answers without any relevant information, relying solely on existing knowledge. Such responses are often inaccurate or even entirely incorrect. Consequently, while LLMs demonstrate potential in code detection, these limitations need to be further addressed.

III. Design Challenges and Solutions

To address these limitations, we propose RuleDroid, an adaptive approach for Android app security detection. RuleDroid leverages LLMs to automatically generate dynamic detection rules based on the latest Android security guidelines, while using static analysis for precise vulnerability assessment. This approach combines the strengths of both LLMs and static analysis, integrating the language understanding and reasoning capabilities of LLMs with the reliability of static detection. As a result, RuleDroid enables dynamic, real-time updates to detection rules without manual maintenance, ensuring the continuous expansion of the rule set. Additionally, by relying on stable, static detection rules for the final assessment (which do not depend on LLM outputs), RuleDroid mitigates issues such as hallucinations and input-size limitations.

Despite these advancements, there remain some technical challenges in implementing RuleDroid. Here, we outline these challenges and present our proposed solutions.

A. Challenges

RuleDroid still needs to overcome some design and implementation challenges to generate reliable static detection rules automatically.

- **Challenge 1:** ***How to select and process the sources for detection rule generation?*** Reliable sources of security requirements and vulnerability descriptions are essential for generating detection rules. The quality of these sources directly impacts the accuracy and reliability of the rules. In addition, different sources may vary in terms of detail, format, and trustworthiness, making it necessary to consider how to process and normalize them effectively.
- **Challenge 2:** ***What rule format should be used for LLM-generated detection rules?*** The generated detection rules will be used in static analysis to detect security issues in apps. Therefore, these rules should be concise, clearly defined, and structured in a format suitable for static analysis to reduce the probability of errors during LLM-based rule generation. Additionally, the rules should be modular and maintainable, allowing for easy updates or extensions as Android evolves.
- **Challenge 3:** ***How to improve the correctness of generated rules?*** Although RuleDroid avoids LLM-related issues during the detection phase, rule generation still relies on LLMs. Consequently, LLMs' inherent instability and risk of hallucinations persist, which can lead to the generation of erroneous rules. If used for detection, such incorrect rules may either fail to function properly or produce false positives. Therefore, incorporating mitigation measures to improve the usability and accuracy of rules is critical.

B. Solutions

To address these challenges, we propose a holistic strategy that forms a complete solution, with details outlined below.

Solution to Challenge 1. In Android security research, official documentation is highly authoritative and accurate. Therefore, we utilize official Android documentation [14] as the source for generating detection rules. This approach offers three main advantages:

- *Authoritative Source.* Maintained by Google, the official documentation undergoes strict internal validation and frequent updates, ensuring accuracy and consistency while avoiding biases that can arise in third-party sources.
- *Up-to-Date Content.* As the Android platform evolves and new vulnerabilities emerge, the documentation reflects these changes promptly, keeping detection rules aligned with the latest security requirements.
- *Comprehensive and Standardized.* The official documentation covers platform security requirements in depth and uses standardized terminology, making it ideal for subsequent automated processing.

To systematically extract security-related information from official documentation, RULEDROID uses a two-step approach. First, an alert statement extraction technique [21] filters and retains only relevant security pages. Second, an LLM identifies and formats the security text blocks. This dual-filtering mechanism effectively removes non-security-related data from large documents while producing well-structured text blocks for subsequent processing.

Solution to Challenge 2. Our primary objective is to generate detection rules. Rather than creating yet another tool from scratch, an extensible existing framework should be integrated. We choose **Semgrep** [22] as RULEDROID's static detection engine, ensuring that all generated rules comply with its format requirements. This choice is motivated by several reasons:

- *Established frameworks are preferable.* LLMs are trained on large-scale corpora and typically lack knowledge of custom-built frameworks. In contrast, mainstream static analysis tools such as CodeQL [23] and Semgrep are well-represented in training data, resulting in higher-quality rule generation when using established frameworks.
- *High compatibility with Android.* Semgrep natively supports key Android development languages (Java, Kotlin) and performs detection without requiring a full build environment.
- *Concise Syntax.* Semgrep's YAML-based rule syntax is concise and intuitive, reducing the likelihood of errors during LLM-based rule generation. In contrast, frameworks like CodeQL use a specialized query language (QL), which in our preliminary tests was more challenging for LLMs to handle without human assistance.
- *Modular rule files.* Semgrep's design allows each rule to be managed in its own independent file, simplifying subsequent dynamic rule updates.

Solution to Challenge 3. Since hallucinations stem from the inherent probabilistic nature of LLMs and are currently difficult to completely eliminate, our focus is on mitigating their impact as much as possible to enhance rule correctness. To mitigate LLM hallucinations, RULEDROID applies two main strategies: Retrieval-Augmented Generation (RAG) and crowdsourcing workflow design.

RAG integrates external knowledge retrieval into an LLM-based generation process, expanding the model's effective "knowledge" beyond its original training data. During rule generation, RULEDROID provides relevant Semgrep rule examples and syntax documentation as well as Android security knowledge to the LLM. This additional context significantly improves the syntactic correctness of generated Semgrep rules and reduces false positives by supplying detailed, domain-specific information. Thus, the LLM is provided with standardized syntax and security knowledge, minimizing errors [17].

The crowdsourcing workflow enhances the LLM's reasoning by decomposing complex tasks into organized stages, allowing the model to iteratively build toward the final result. Instead of relying on a single, lengthy prompt, RULEDROID orchestrates multiple LLMs across different stages, where the intermediate outputs are sequentially passed as inputs to subsequent steps. This structured workflow design enables the model to focus deeply on each subproblem, substantially improving the overall accuracy and granularity of its analysis, while mitigating hallucinations arising from excessive task complexity [15].

Another advantage of the workflow design is its ability to assign each stage to a dedicated LLM, each configured with a task-specific RAG knowledge base and selected for its respective strengths. For example, text extraction can be handled by a model optimized for linguistic processing, while logical reasoning can be delegated to one specialized in analytical inference. By aligning each subtask with the most suitable LLM within the workflow, RULEDROID maximizes both efficiency and accuracy, while minimizing hallucinations. Compared with autonomous decision-making agents, this modular workflow architecture ensures controllable outputs throughout the entire detection rule generation process, thereby maximizing stability and maintainability.

IV. DESIGN OF RULEDROID

This section presents the detailed design of RULEDROID. At a high level, the complex processes of rule generation and vulnerability detection are organized into a structured workflow comprising five main phases. Each phase corresponds to a specific subtask within the overall detection pipeline. An overview of the entire workflow is shown in Figure 1.

Phase 1: Data Acquisition. This step collects comprehensive and up-to-date security knowledge to support rule generation. It crawls official documentation and other relevant resources to construct multiple task-specific knowledge bases.

Phase 2: Document Preprocessing. This step transforms raw documents into structured and relevant inputs for the LLM. It removes noise, segments long texts, and extracts security-related content blocks from the crawled data.

Phase 3: Detection Rule Generation. This step selects the security guidelines applicable to static detection rules and generates detection rules based on the guidelines. Then, syntax checking and iterative repair are performed to ensure the enforceability of the rules.

Phase 4: Rule Refinement and Optimization. This step improves the robustness and effectiveness of the rule set. It enhances the quality of generated rules and removes redundancy. It also classifies each rule into severity levels, helping users focus on high-risk issues.

Phase 5: App Security Analysis. This step applies the generated rules to input APKs by performing decompilation and static analysis. It produces structured results to support large-scale vulnerability auditing.

Different LLM instances were used for each task in the process. The corresponding models and their associated RAG knowledge bases are summarized in Table I, and the model selection is discussed in Section V-A.

A. Phase 1: Data Acquisition

The goal of this phase is to gather official Android documentation and relevant security materials for RULEDROID, and to build multiple knowledge bases that support the subsequent RAG operations.

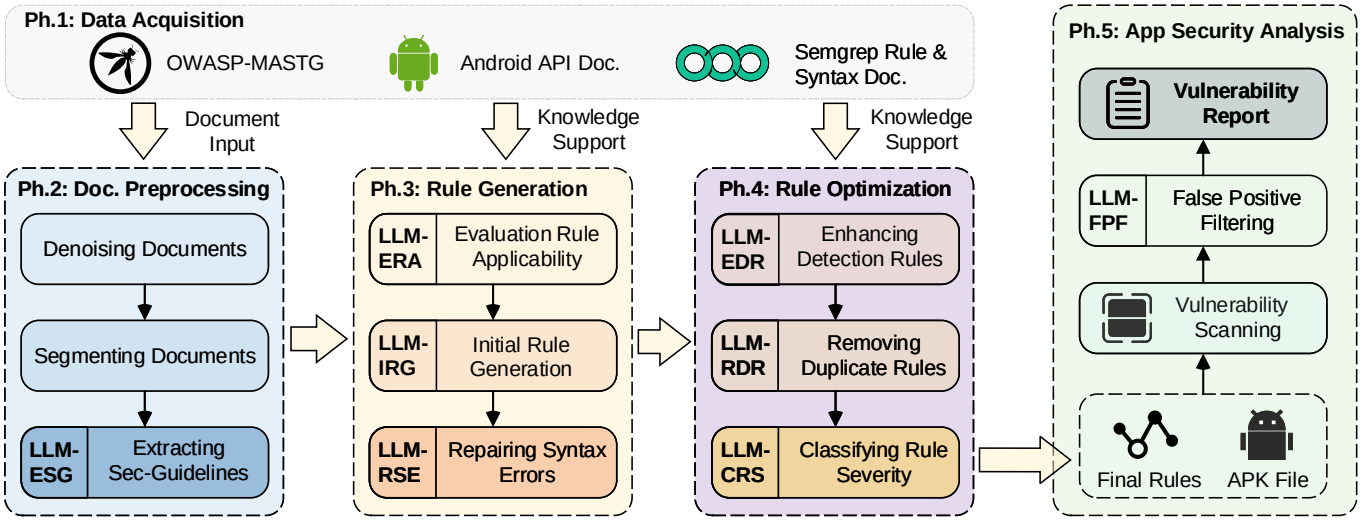

Fig. 1: Overall workflow of RULEDROID.

TABLE I: Configurations of the LLMs used in RULEDROID.

Custom LLM	Operational Task	Model	RAG			
			Droid-API	SG-Rule	SG-Doc	OWASP
LLM-ESG	§IV-B Extracting Security Guidelines	GPT-5	○	○	○	●
LLM-ERA	§IV-C Evaluating Rule Applicability	o4-mini	○	●	○	●
LLM-IRG	§IV-C Initial Rule Generation	GPT-5	●	●	●	●
LLM-RSE	§IV-C Repairing Syntax Errors	GPT-5	○	●	●	○
LLM-EDR	§IV-D Enhancing Detection Rules	o4-mini	●	●	●	●
LLM-RDR	§IV-D Removing Duplicate Rules	DeepSeek-V3.2	○	○	○	○
LLM-CRS	§IV-D Classifying Rule Severity	DeepSeek-R1	○	●	○	●
LLM-FPF	§IV-E False Positive Filtering	GPT-5	○	○	○	●

(1) Obtaining Security Documentation. To collect comprehensive security-related knowledge for RAG, we use BeautifulSoup4 [24] to crawl a set of authoritative sources, including the official Android documentation [14], the OWASP Mobile Application Security Testing Guide (MASTG) [25], Semgrep documentation [22], and its sample rule repositories [26].

During the crawling process, RULEDROID adopts the alert statement extraction method proposed by [21], retaining only pages containing explicit warning information. This filtering step significantly improves data relevance, removes noise, and prevents unnecessary resource consumption. While this intentional filtering reduces the absolute completeness of the raw text, it significantly improves the signal-to-noise ratio, thereby enhancing the quality and precision of the auxiliary context retrieved by the RAG system for rule generation.

(2) Building Knowledge Bases. Most of the crawled documents (such as the Android API reference under the `/reference` URL path) will be used to build multiple specialized knowledge bases. These knowledge bases employ an embedding-based vector retrieval mechanism: raw text is converted into vector embeddings, and semantic similarity is measured by comparing distances between vectors. During query time, document fragments are retrieved as context for the LLM, enabling more accurate and relevant responses.

For the RAG setup, RULEDROID uses PostgreSQL with PGVector for efficient vector indexing. Index uses the Hierarchical Navigable Small World (HNSW) algorithm [27], which is known for fast retrieval and high recall in large-scale searches. PostgreSQL is used solely for vector retrieval, while the original metadata of the vectors is stored in MongoDB.

In vector representations, there is often a trade-off between content length and semantic richness. To mitigate this, we adopt a multi-vector mapping strategy, where each data entry is encoded into multiple vectors to preserve both completeness and semantic detail. During retrieval, the system first recalls relevant vectors, then uses their IDs to query MongoDB for the corresponding original content. If multiple vectors map to the same source, their results are merged, and the highest vector score is taken as the final relevance score. Meanwhile, it utilizes OpenAI's `text-embedding-3-large` model [28] to ensure high-quality semantic representations. Because different steps of RULEDROID require different types of knowledge, four independent knowledge bases are established as needed:

- **RAG-Droid-API**: Android API Documentation. This knowledge base contains all official Android API pages [14] that include warnings, covering function descriptions, class definitions, and usage examples for code-related queries.
- **RAG-SG-Rule**: Semgrep Android Rule. This knowledge base consists of 107 Android-specific detection rules from

Semgrep's official documentation [26] and the open-source community [29], serving as references for rule formats and pattern design, rather than a baseline for adapting existing rules. This ensures that the generated rules are new and their logic is derived exclusively from the Android documentation.

- **RAG-SG-Doc**: Semgrep Syntax Documentation. This knowledge base encompasses official syntax references and examples from Semgrep [22], guiding accurate rule generation and repair.
- **RAG-OWASP**: This knowledge base is derived from the OWASP MASTG [25], covering typical attack surfaces and security testing techniques to enhance the LLM's analytical capabilities.

B. Phase 2: Document Preprocessing

RULEDROID selected two types of documents as security references for generating static analysis rules. The first is the content under the `/privacy-and-security` section of the official Android documentation, which covers the core security guidelines for the Android platform. The second is the `Document` section of the OWASP MASTG, which supplements security issues related to the use of Java and cryptographic algorithms in apps.

Next, RULEDROID further processes these documents to extract clean and relevant data. These HTML-format files contain not only a large volume of HTML tag noise but also non-security content. To address this, RULEDROID follows a three-step preprocessing procedure: document denoising, document segmentation, and extracting security guidelines. Finally, this preprocessing yields isolated, valid content segments, preventing interference in subsequent tasks.

(1) Denoising Documents. RULEDROID first uses the tomd library [30] to convert HTML files into Markdown, a process that removes most HTML tags while preserving structural elements such as headings and code blocks, ensuring that semantic information is fully retained in a clean, natural language format. It then further cleans the output by deleting tags like `<li>`, `<i>`, and `<a>` (while retaining their text). This specific cleanup is necessary because, in certain instances, the tomd library successfully converts the structural formatting but leaves the original HTML tags behind as non-semantic noise. Furthermore, RULEDROID removes `<img>` tags and associated content, and eliminates HTML-specific markers like ` `. The result is a Markdown-format document containing only core textual information, providing a clean and structured input for the next steps.

(2) Segmenting Documents. Due to the input length limits of the LLM, RULEDROID segments documents that exceed the model's input capacity. To maintain logical consistency, RULEDROID leverages the heading information preserved during denoising, splitting at heading structures near the average token position. If a document still exceeds the limit after initial segmentation, it is further subdivided until all segments comply with the model's input restrictions.

(3) Extracting Security Guidelines. A single document may contain multiple security guidelines or none at all. Hence, RULEDROID employs the LLM-ESG to identify and extract potential security guidelines in a specified format. Each extracted text block corresponds to exactly one security guideline (as shown in Listing 1), thereby ensuring each valid guideline is isolated from the others. Since detection rules primarily focus on guidelines directly impacting security, RULEDROID tags those labeled as "BEST PRACTICES IN DEVELOPMENT" and excludes them from rule generation. This step ensures that only security-relevant content is retained.

```
1 ====== Guideline-1
2 name: Cleartext Communications
3 type: Safety Requirements and Specifications
4 info:
5 Allowing cleartext network communications in
   an Android app means that anyone monitoring
   network traffic can see and manipulate the
  data that is being transmitted. This is a
  vulnerability if the transmitted data
  includes sensitive information such as
  passwords, credit card numbers, or other
  personal information...
6 ====== Guideline-2
7 name: xxxxx
8 type: Best Practices in Development
9 ...
```

Listing 1: Example of text block.

Furthermore, although extracting structured guidelines inherently causes some information loss, RULEDROID mitigates this by storing the complete original documentation in the RAG knowledge base. During the actual rule generation phase, the RAG mechanism retrieves this full context to effectively supplement any details that might have been lost during the initial extraction step.

C. Phase 3: Detection Rule Generation

After extracting text blocks that contain security guidelines, RULEDROID generates Semgrep static detection rules in the following three steps.

(1) Evaluating Rule Applicability. Static detection rules have inherent limitations and cannot address certain security issues, such as runtime vulnerabilities or highly customized attack vectors. To avoid producing invalid or erroneous rules that could lead to false positives and reduce overall detection efficiency, RULEDROID uses LLM-ERA to determine whether each security issue falls within the scope of static detection. This filtering process improves the accuracy and practicality of subsequent rule generation.

Because the LLM is responsible for making these judgments, and the judgment task is essentially a binary (yes/no) decision, the potential hallucinations of the LLM could severely impact the results. To enhance reliability, RULEDROID employs a multi-round voting mechanism. Specifically, RULEDROID will conduct five rounds of voting, and text blocks that are judged as "unsuitable" in four or more rounds will be discarded. This approach provides high accuracy and maintains efficiency by minimizing the impact of LLM hallucinations.

(2) Initial Rule Generation. After confirming that a text block is suitable for static detection, RULEDROID passes its

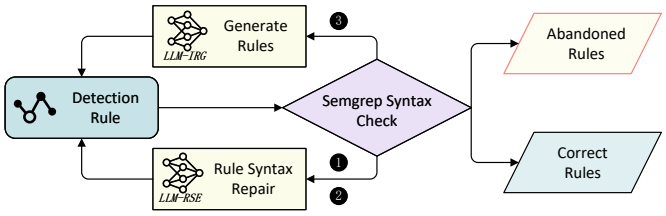

Fig. 2: Workflow of repairing syntax errors.

content to the LLM-IRG for rule generation. The LLM-IRG then produces one or more Semgrep detection rules tailored to the specific security issue. Examples of the initial rules generated are shown in Listing 2.

```
rules:
 - id: cleartext-communication-android
 languages:
    - xml
 message: |
    Cleartext traffic is permitted in ...
 paths:
    include:
      - '**/AndroidManifest.xml'
 patterns:
    - pattern: <application ... android:
usesCleartextTraffic="true" ... />
    - pattern-not: <application ... android:
usesCleartextTraffic="false" ... />
 severity: WARNING
 - id: cleartext-communication-network-
config-base
    ...
 - id: cleartext-communication-network-
config-domain
    ...
```

Listing 2: Example of initial rule.

(3) Repairing Syntax Errors. Despite efforts to minimize errors, some generated rules may contain syntax issues due to LLM hallucinations. To address this, RULEDROID performs a syntax check on each rule and repairs any errors encountered. First, it uses Semgrep's built-in self-check function to detect syntax problems and retrieve the associated error messages. Next, the erroneous rule code, error messages, and the original text block are passed to LLM-RSE for syntax correction.

As illustrated in Figure 2, RULEDROID adopts a multi-stage repair process for rules with syntax errors: ❶ LLM-RSE is first used to repair the rule. If errors persist, ❷ LLM-RSE is applied repeatedly until the errors are resolved; if the problem remains after multiple rounds, then ❸ LLM-IRG is used to regenerate the rule, followed by further rounds of repair via LLM-RSE if necessary. If syntax errors continue to persist after these steps, the system deems the corresponding security guideline incapable of yielding a valid detection rule and discards it. In practice, very few rules are abandoned, as shown in Table XIV of Appendix B. Also, the number of repair rounds is empirically set to 5. The details of the experiments regarding the repair round settings are as follows:

We conducted repair-round experiments on the *Repairing Syntax Errors* and *Enhancing Detection Rules* tasks using a small-scale document dataset. The experiments involved LLM-RSE and LLM-EDR, which evaluated the effectiveness of the generated rules across different repair rounds by measuring their recall on the DroidCVE++ benchmark. To reduce the impact of LLM randomness, each experiment was repeated five times, and the average result was reported. As shown in Figure 3, the results stabilize after about 5 rounds. Therefore, we set the number of repair rounds to five in our evaluation to balance performance and efficiency.

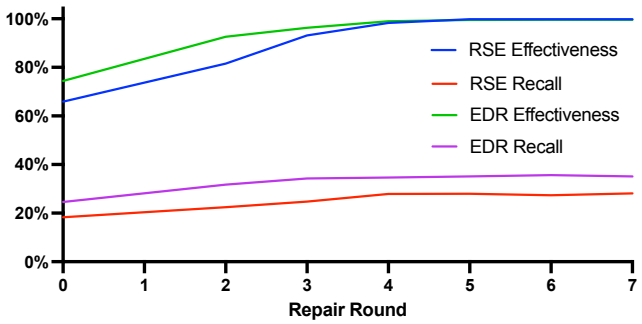

Fig. 3: Effect of LLM-RSE repair rounds on rule quality.

D. Phase 4: Rule Refinement and Optimization

To further enhance the quality of the rules, improve detection accuracy, reduce the false positive rate, and eliminate duplicate rules targeting the same security issue, RULEDROID performs additional optimization after rule generation. This optimization step is divided into three components: enhancing detection rules, removing duplicate rules, and classifying rule severity.

```
rules:
 - id: cleartext-communication-android
   languages:
    - xml
   message: |
      Cleartext traffic is permitted in ...
   paths:
    include:
     - '**/AndroidManifest.xml'
   patterns:
    - pattern-either:
       - pattern:  <application ... android:
usesCleartextTraffic="true" ... />
       - pattern: |
         <application ...
         android:usesCleartextTraffic="true"
         ... />
       - pattern-regex:  <application[^>]*
android:usesCleartextTraffic=["']true["']
    - pattern-not-regex:  <application[^>]*
android:usesCleartextTraffic=["']false["']
   severity: WARNING
   ...
```

Listing 3: Example of enhanced rule.

(1) Enhancing Detection Rules. Transformer-based LLMs may divide their attention when simultaneously generating multiple rules for a single security issue [13], preventing the model from producing the optimal version of each rule and

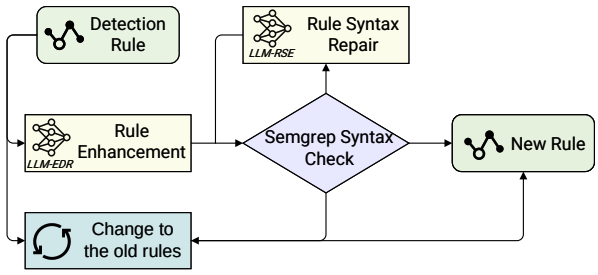

Fig. 4: Workflow of rule enhancement.

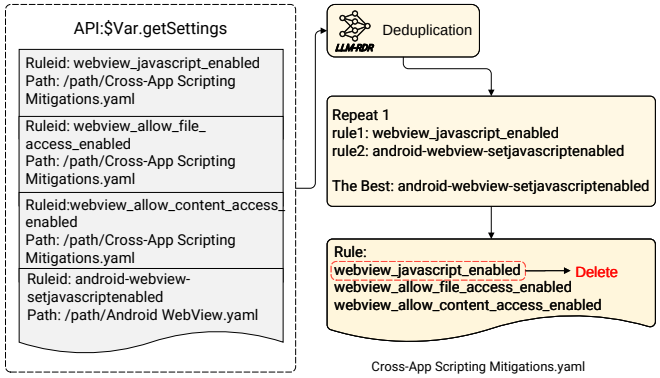

Fig. 5: Workflow of removing duplicate rules.

potentially leading to incomplete detection coverage. To address this, RULEDROID splits files containing multiple Semgrep rules and processes each individually using the LLM-EDR. By focusing on one rule at a time, the LLM can dedicate its full attention to optimizing that specific rule, enabling it to more effectively address the corresponding security issue and reduce false positives.

The enhanced rules are shown in Listing 3. Compared with the initial rules, the enhanced versions demonstrate more robust matching capabilities (supporting multi-line expressions and adaptive spacing adjustments) as well as a more flexible exclusion mechanism that employs regular-expression-based exclusion criteria.

However, expanding rules can introduce new syntax errors. As illustrated in Figure 4, these enhanced rules are passed again to the LLM-RSE. If repeated repair attempts fail for a given guideline, RULEDROID reverts to using the Initial rule rather than discarding the guideline altogether. We set the maximum number of repair attempts to 5 rounds. The detailed evaluation experiments can be found in Section IV-C.

(2) Removing Duplicate Rules. While RULEDROID intentionally generates multiple distinct rules for the same sensitive API to capture various vulnerable code patterns, different pages in the Android documentation may repeatedly mention the same security issue, leading to duplicate rule generation. Additionally, two related but distinct issues can end up with overlapping detection scopes after enhancement, thus being considered duplicates. To avoid wasting resources and skewing scan results, RULEDROID identifies and removes redundant rules.

First, RULEDROID checks for duplicate rule names and appends sequential suffixes to any duplicates to avoid conflicts. Next, rules suspected of duplication are submitted to the LLM-RDR. RULEDROID establishes the following criteria for identifying potential duplicates:

- *Rules for Java and Kotlin Files:* Potential duplication is assessed by examining the APIs used in each rule's matching patterns. If two rules target the same API, they are considered duplicates.
- *Rules for XML Files:* RULEDROID applies a Locality-Sensitive Hashing (LSH) algorithm to measure similarity in matching patterns. If the similarity exceeds a predefined threshold[1], the rules are deemed duplicates.

[1]After experimental evaluation and balancing robustness and precision, the LSH parameter configuration is set as follows: Shingles length of 5, MinHash signature length of 256, and a matching threshold of 0.5.

As illustrated in Figure 5, after initially screening out potential duplicate rules, the LLM-RDR refines the selection process by identifying truly redundant rules, determining the optimal candidate within each duplicate group, and discarding the rest to maintain a streamlined and consistent rule set.

To select the optimal detection rule, LLM-RDR applies a hierarchical evaluation criterion. The primary metric is the breadth of vulnerability coverage—rules that cover a wider range of variants (i.e., with more comprehensive `patterns` fields in Semgrep) are prioritized. If multiple rules offer comparable coverage, preference is given to those with stronger false-positive filtering capabilities, as indicated by the `pattern-not` field. When rules are still comparable, auxiliary information is considered, favoring those with complete vulnerability descriptions (`message` field) and detailed remediation guidance (`fix` field).

(3) Classifying Rule Severity. While the rules generated by RULEDROID cover all security issues identified in the official Android documentation, their practical impact may vary. To support effective prioritization, RULEDROID assigns a severity level to each rule, allowing users to focus on high-risk issues while maintaining a structured view of the overall scan results.

Following Semgrep's classification system, RULEDROID categorizes each rule into one of three severity levels: `ERROR`, `WARNING`, or `INFO`. To define these categories, we systematically analyzed vulnerability classification methods from the official Semgrep rule repository and other open-source Android Semgrep examples, then formulated the following specific criteria:

- `ERROR`: Direct security issues, such as hardcoded keys, deprecated APIs, weak cryptographic algorithms.
- `WARNING`: Potential security risks, such as incomplete configurations and insecure random number generation.
- `INFO`: Checks for security requirements, such as sensitive permission calls.

These guidelines are incorporated into the LLM-CRS via a structured system prompt, which facilitates automatic severity labeling for detection rules. By following this tiered approach, RULEDROID maintains alignment with industry standards and assists users in rapidly identifying and addressing the most critical security issues.

E. Phase 5: App Security Analysis

After generating the detection rules, RULEDROID launches its vulnerability detection workflow. The target APK is first decompiled with JADX [31] (an Android decompiler), yielding analyzable Java code and XML configuration files. Subsequently, static vulnerability analysis is conducted using the Semgrep engine. This process is organized into four steps:

(1) Syntax Structure Modeling. The source files are parsed into an Abstract Syntax Tree (AST) based on the programming language, thereby constructing a comprehensive semantic representation of the program.

(2) Pattern Matching Detection. Pattern matching is performed by processing the rule's `pattern` field, which is parsed into a Pattern AST. This Pattern AST then traverses the entire source code AST, attempting structural alignment with each subtree. Matching requires consistency in node types, keywords, and attributes, while metavariables (e.g., `$KEY`, `$VALUE`) enable dynamic binding to accommodate patterns with variable content yet stable structure.

(3) False Positive Filtering. Following initial rule matching, a series of filtering mechanisms further refine the results. We construct a negative pattern list using Semgrep's `pattern-not` operator, which explicitly defines code structures that should be excluded. In addition, contextual constraints (e.g., `pattern-inside`, `pattern-either`) specify the valid code context or extend pattern expressiveness, while regular expressions (`pattern-regex`) validate the content of bound metavariables. Only nodes satisfying all matching and filtering conditions are retained as valid findings.

```
"com.xxxxxx.xxxxxxmessenger.apk": [
 { "check_id": "hardcoded-crypto-secrets",
   "end": {"col": 94, "line": 18, "offset":
655},
   "extra": {
     "lines": "SecretKeySpec secretKeySpec =
 new SecretKeySpec(\"RfTjxxxxxx\".getBytes()
, \"AES\");",
     "message": "Hardcoded crypto secret
detected",
     "metavars": {"$KEY": {
       "abstract_content": "RfTjxxxxxx",
       "end": {"col": 74, "line": 18, "
offset": 635},
       "start": {...}}},
     "severity": "ERROR",    ...},
   "path": "/apppath/xxx.java",
   "start": {...},
   "vul_check":"Vul:True Reason:XXXXX"
  }]
```

Listing 4: Example of detection report (part).

To further reduce false positives, RULEDROID invokes LLM-FPF to reassess the rule-based results. It supplies the LLM with matched code snippets, contextual information, and descriptions of the detected vulnerability types. LLM-FPF evaluates each finding and discards those likely to be false positives, ensuring that only credible results appear in the final report.

Although the incorporation of the LLM-FPF introduces a degree of uncertainty into the final detection reports, the stability evaluation of RuleDroid in Section V-B and the ablation study results in Section V-C demonstrate that this impact on result certainty is relatively minor. Meanwhile, the benefits of the LLM-FPF to the overall detection efficacy significantly outweigh its potential drawbacks.

(4) Detection Report Generation. For each successfully matched vulnerability instance, a structured record is generated that includes metadata such as code location, rule identifier, and severity level. All detection results are then consolidated into a standardized JSON report, as shown in Listing 4 (some sensitive fields are redacted).

F. Remarks

Regular Usage and Rule Update. After the rule initialization phase, regular use of RULEDROID begins in Phase 5, the vulnerability detection phase. There is no need to regenerate rules frequently.

Furthermore, when Android OS security requirements change, RULEDROID can update its rule set and knowledge base accordingly. During the update process, it does not need to regenerate all rules or rebuild the entire knowledge base. Instead, only the modified parts are processed. During the initial crawling phase, RULEDROID records the ETag value [32] of each page (all target websites support ETag). A rule or knowledge entry is updated only if the corresponding page content changes, as indicated by a changed ETag value.

This mechanism demonstrates the advantage of using RAG over model fine-tuning. RULEDROID can efficiently remove outdated knowledge from PostgreSQL and MongoDB and replace it with up-to-date content, without incurring the overhead of re-training the model.

Validation Strategy and Fault Tolerance. Throughout the RULEDROID pipeline, validating LLM outputs is crucial. Enabled by the modular design of our workflow, we can employ distinct validation strategies tailored to the specific characteristics of each stage. For some stages, such as rule generation, repair, and enhancement, the pipeline strictly enforces automated validation by executing the outputs through the Semgrep engine to ensure syntactic correctness. However, for certain intermediate parsing steps that lack an objective ground truth for semantic correctness, the system relies exclusively on structural format verification. To ensure the reliability of these steps, we conducted manual sampling and evaluations, confirming consistently high accuracy. Furthermore, our manual analysis indicates that the overall pipeline exhibits high fault tolerance; minor semantic deviations in early steps almost never propagate to cause critical errors in the final generated rules, as downstream validations naturally filter out invalid logic.

V. EVALUATION AND RESULTS

In this section, we conduct a comprehensive evaluation of RULEDROID by comparing it against various static analysis tools and state-of-the-art (SoTA) LLMs across three benchmarks. We also examine RULEDROID's performance on real-world Android apps to assess its effectiveness at identifying vulnerabilities in practical settings. We focus on the following five research questions.

RQ1: *How stable and correct is* RULEDROID *in rule generation?*

RQ2: *How does each component of* RULEDROID *contribute to the overall effectiveness of vulnerability detection?*

RQ3: *Compared with traditional Android SAST tools, how does* RULEDROID *perform in vulnerability detection?*

RQ4: *Compared with direct LLM-based methods, how does* RULEDROID *perform in vulnerability detection?*

RQ5: *How does* RULEDROID *perform on real-world apps?*

A. Experimental Setup and Benchmarks

Implementation and Setup. RULEDROID is implemented in Python and comprises over 2,600 lines of code. All experiments were conducted on a server running Ubuntu 22.04.5 LTS with an AMD EPYC 9654 CPU (96 cores) and 1 TB of RAM.

Model Selection. As described in Section III-B, the workflow design of RULEDROID allows different LLMs to be applied at each step. To determine the optimal configuration, we evaluated two closed-source advanced models (GPT-5 and o4-mini) and two open-source advanced models (DeepSeek-V3.2 and DeepSeek-R1) across all LLM-related components. Details of the evaluation setup and results are provided in Appendix B. Note that, RULEDROID is designed to support different LLMs and is not tied to any specific provider. It can integrate with both proprietary and open-source models. However, to achieve the best results for RULEDROID, the most effective model combinations were used at each step of the evaluation. Table I lists the selected model for each step.

Time and Cost for Rule Generation. The initial rule generation with RULEDROID takes about 1.2 hours and costs approximately $32, processing 13.9 million input tokens and 1.73 million output tokens. RULEDROID significantly reduces the overall time and cost compared with manual rule creation and maintenance. In contrast, an experienced security expert would typically spend several days reviewing and extracting rules from hundreds of pages of documentation. For subsequent updates (see Section IV-F), RULEDROID only regenerates rules affected by new changes in the security guidelines.

App Analysis Efficiency. RULEDROID, using the generated rules and analyzing a single app on a single thread, takes about 16-60 seconds, including APK unpacking, depending on the app's size. When using 50 threads in parallel, the underlying SAST engine can analyze 1,000 apps in under 30 minutes. When accounting for the complete pipeline, including LLM-FPF, RULEDROID fully processes around 250 apps in 30 minutes.

Benchmarks. We used the following three benchmarks for comparative evaluation:

- ***MASTG&PIVAA Benchmark.*** OWASP's MASTG [33] and High-Tech Bridge's PIVAA [34] each offer a limited set of vulnerable Android apps. We merged them into one benchmark to enhance evaluation diversity and coverage. This benchmark contains 52 vulnerability cases.
- ***Ghera Benchmark.*** Ghera [35] covers a variety of vulnerability types by providing both vulnerability cases and security cases for each category. This design makes Ghera particularly useful for evaluating false positive rates, with the benchmark comprising 59 vulnerability cases and 59 security cases.
- ***DroidCVE++ Benchmark.*** While synthetic benchmarks like MASTG&PIVAA and Ghera are valuable, a dataset reflecting real-world app vulnerabilities is also necessary. To address this need, we enhanced the original VulsTotal-CVE benchmark [36] to create the DroidCVE++ Benchmark, which incorporates actual app vulnerabilities for a more realistic evaluation. This benchmark encompasses 278 vulnerability cases (excluding our obtained CVEs), where the newly added cases are sourced from public CVEs outside the original dataset that provide sufficiently clear reports to accurately pinpoint the exact vulnerability locations.

Note that, none of the benchmark code appears in the RAG knowledge base. Although both the RAG-OWASP and MASTG benchmarks were derived from OWASP Mobile Application Security (MAS), they were sourced independently. The RAG repository does not contain any benchmark code. The RAG documents were extracted from the textual content of OWASP-MASTG [25], while the benchmark apps were taken from OWASP/MASTG-Hacking-Playground [33].

The improvements introduced in the DroidCVE++ Benchmark are as follows. The original CVE-based benchmark included only vulnerable APKs and CVE type labels, making it difficult to verify whether detection results truly corresponded to specific CVE instances. DroidCVE++ addresses this limitation by manually extracting vulnerable code files based on the descriptions in CVE entries. Each sample now includes precise source files and their corresponding file paths, enabling verifiable vulnerability locations. CVE entries with unclear or incorrect descriptions have been excluded. In addition, the benchmark's coverage has been expanded from 34 to 43 vulnerability types, encompassing 278 CVEs.

During this extension process, we discovered issues with the original VulsTotal-CVE benchmark, including version mismatches in sample APKs and inaccuracies in certain CVE entries. These problems were reported to the original development team, and corresponding corrections were applied to the DroidCVE++ benchmark, thereby ensuring the dataset's accuracy.

Evaluation Metrics. Since the Ghera benchmark includes both vulnerability and benign cases, metrics such as false positives (FP), true negatives (TN), and false positive rate (FPR) could be computed. In contrast, for MASTG&PIVAA and DroidCVE++, which include only vulnerability cases, we measured true positives (TP), false negatives (FN), and recall. Across all three datasets, we also recorded the number of vulnerability reports (#Rep). To further evaluate precision on these two datasets, we manually inspected the extra vulnerability reports generated outside the benchmark. For each tool, we randomly sampled 50 extra reports, or inspected all if the total was under 50. Because this manual inspection only evaluates the predicted positives, the total number of TN remains unknown, making the FPR unfeasible to compute. Consequently, we calculated the False Discovery Rate (FDR) instead.

TABLE II: Statistics of rule generation.

Instance #	Total Rules	ERROR	WARNING	INFO
RuleDroid$_1$	406	201	98	107
RuleDroid$_2$	398	186	100	112
RuleDroid$_3$	415	198	103	114

TABLE III: Semantic consistency of generated rules.

Instance #	Consistent	Inconsistent		Cons. Rate
		Valid	Invalid	
RuleDroid$_1$	96	2	2	96.0%
RuleDroid$_2$	93	5	2	93.0%
RuleDroid$_3$	95	4	1	95.0%
Overall	284	11	5	94.7%

B. RQ1: Rule Generation Stability and Correctness

To assess the stability and correctness of RuleDroid in generating detection rules, we executed the system three times. Each run took approximately 1.2 hours and produced a distinct set of rules. We denote these sets as RuleDroid$_1$, RuleDroid$_2$, and RuleDroid$_3$. All three runs utilized identical environments and model configurations. We compared these sets to evaluate the stability and correctness of RuleDroid.

Rule Generation Stability. The results of these runs are listed in Table II. As shown, the number of generated rules is roughly consistent across runs. It is important to note that variations in the number of rules do not necessarily imply that some checks are missed. Specifically, some issues are split into multiple rules for detection, while others are consolidated into a single rule. For example, in RuleDroid$_1$, the weak hash issue is divided into weak_hash_md5, weak_hash_sha1, and weak_hash_other, whereas in RuleDroid$_3$, a single weak_hash rule is used to address the same issue. Variations in runtime are primarily attributed to differences in the number of rules. These differences, in turn, cause varying numbers of LLM requests and fluctuations in response times.

Although the number of generated rules remains stable, this metric alone does not reflect actual detection stability. To evaluate detection consistency, we tested RuleDroid$_{1\text{-}3}$ across three benchmark datasets. Table V shows that all three rule sets achieve similar outcomes across key metrics. This indicates that RuleDroid exhibits high stability from the initial rule generation phase to final vulnerability detection.

Rule Generation Correctness. To evaluate the correctness of the generated rules, we manually assessed their semantic consistency with the corresponding source documents. We consider a rule semantically correct if its detection logic reflects the security requirement described or directly implied by the source document. We randomly sampled 100 rules from each of the three generated rule sets for a total of 300 rules. Two team members with expertise in Android security and static analysis evaluated each rule alongside its original document fragment. They independently labeled the rules. The evaluators then measured inter-rater agreement and resolved disagreements through discussion.

The evaluators labeled a rule as Consistent if the source document explicitly described or directly implied its detection target, API, configuration, risky condition, or contextual constraint. They labeled all other rules as Inconsistent. Table III presents these results. Overall, 284 of the 300 sampled rules received a Consistent label, yielding a consistency rate of 94.7%. This indicates that most of the generated rules successfully capture the requirements from their source documents rather than merely passing Semgrep syntax checks. We further classified each Inconsistent rule. We labeled a rule as Valid if it detects a reasonable security concern not described in the official source document. We labeled the remaining rules as Invalid. Only five rules received an Invalid label. This low number demonstrates that truly invalid rules are rare in our generated sets.

We further inspected the Inconsistent rules. The primary cause is that LLM-based rule generation relies on probabilistic semantic associations rather than strict traceability between rule constraints and source documents. For example, one Valid case involves a rule detecting enabled WebView debugging. Although the source document discusses general debugging issues, it does not explicitly mention WebView debugging. We verified this configuration as a legitimate security concern [37]. In contrast, an Invalid case produced an erroneous rule checking for hardcoded packages in a PendingIntent. The source document focuses on mutable or replay vulnerabilities in a PendingIntent and recommends setting the component or package fields to mitigate the risk. However, the generated rule incorrectly flags the `setPackage()` method as a risk.

Many factors influence practical vulnerability detection beyond the initial correctness of the generated rules, although rule correctness provides a necessary foundation. Section V-D provides a detailed analysis of the detection results.

Short Answer to RQ1

RuleDroid demonstrates high stability across multiple executions during rule generation, leading to consistent vulnerability detection performance. Regarding rule correctness, the vast majority of generated rules accurately capture the security requirements in their source documents.

C. RQ2: Ablation Experiments

To evaluate the effectiveness of each step in the workflow design of RuleDroid, as well as the contribution of the RAG knowledge bases, we conducted a series of ablation studies. The experiments were performed across all three benchmarks to assess the quality of the generated detection rules.

As shown in Table IV, BaseLLM exhibits poor performance across all three benchmarks, indicating that directly relying on LLMs for rule generation is ineffective. Furthermore, the comparative results without RAG and BaseRAG collectively demonstrate that both the RAG mechanism and the structured workflow design play a crucial role in the overall detection capability. Beyond these baselines, each internal component contributes meaningfully to the overall performance. LLM-RSE is particularly important for rule quality. Its removal leads to a 40.65% drop in recall (from 90.29% to 49.64%) on DroidCVE++, primarily due to syntax errors in the generated rules, which prevent correct execution and result in missed vulnerabilities. More broadly, all components are essential, as

TABLE IV: Ablation experiments.

Step	Ghera		MASTG&PIVAA		DroidCVE++	
	F1	F1↓	Recall	Recall↓	Recall	Recall↓
(w/o) ESG	64.44%	-12.48%	61.53%	-19.23%	63.31%	-26.98%
(w/o) ERA	71.70%	-5.22%	73.08%	-7.68%	83.81%	-6.48%
(w/o) RSE	58.70%	-18.22%	50.00%	-30.76%	49.64%	-40.65%
(w/o) EDR	70.71%	-6.21%	65.38%	-15.38%	67.99%	-22.30%
(w/o) RDR	74.29%	-2.63%	78.84%	-1.92%	86.68%	-3.61%
(w/o) FPF	73.87%	-3.05%	76.92%	-3.84%	89.21%	-1.08%
(w/o) RAG	68.04%	-8.88%	65.38%	-15.38%	65.11%	-25.18%
BaseRAG	58.33%	-18.59%	48.08%	-32.68%	43.17%	-47.12%
BaseLLM	53.33%	-23.59%	40.38%	-40.38%	38.13%	-52.16%
RuleDroid$_1$	76.92%	0.00%	80.76%	0.00%	90.29%	0.00%

↓: Difference in compared to RULEDROID$_1$.
BaseRAG: BaseLLM with the support of a RAG knowledge base.
BaseLLM: Generates rules directly using an LLM.

removing any single step causes a notable decline in recall across the datasets.

The evaluation results on the Ghera benchmark demonstrate the effectiveness of LLM-FPF in reducing false positives. Specifically, integrating LLM-FPF reduces the FPR by 10.17% (from 18.64% to 8.47%) and increases the F1 score by 3.05% (from 73.87% to 76.92%).

Short Answer to RQ2

Ablation results show that both the stepwise reasoning in RULEDROID's workflow process and the use of the RAG-based knowledge bases play a key role in improving the quality of generated detection rules.

D. RQ3: Comparison with Android SAST Tools

This section presents a comprehensive comparison between RULEDROID's detection rules and seven representative Android SAST tools: SPECK [18], Marvin [19], AndroBugs [38], QARK [39], MobSF [1], AUSERA [3], and APKHunt [2]. It is worth noting that SPECK's detection rules are also derived from Google's security guidelines but are manually constructed. These tools reflect the current state of static analysis for Android apps. The goal of this comparison is to evaluate RULEDROID's effectiveness in detecting vulnerabilities across diverse and practical scenarios.

Some analysis tools are excluded for two main reasons. First, certain frameworks (e.g., CodeQL [23]) require a complete project environment and rely heavily on customized rules, making standardized evaluation difficult. Second, some tools are not designed for app vulnerability detection and are therefore not comparable to rule-based detection tools. For instance, FlowDroid [40] focuses on data-flow analysis, while others target malware detection.

Rule and Type Coverage. To evaluate the coverage of vulnerability types, we adopt two classification standards: the scheme proposed by Zhu et al. [36] and the OWASP Mobile Application Security Weakness Enumeration (MASWE) [41]. The detection rules of RULEDROID and the seven baseline tools are mapped to the 67 and 116 categories defined by these standards, respectively. The mapping process varies by tool. For AndroBugs [38], we obtained rule counts and types from its official documentation. For MobSF [1], which only provides category names, we extracted detailed rule information from its source code. Similar strategies were applied to the remaining tools to ensure a fair and consistent comparison.

Table VI reports the number of vulnerability types covered by each tool, the number of rules mapped to these categories, and the number of rules addressing issues beyond the defined classifications. Detailed statistics are provided in Table VII. The results show that RULEDROID achieves the broadest type coverage among all tools, demonstrating that rules derived from official Android security guidelines can effectively address a wide range of practical security issues.

It is worth noting that although MobSF includes up to 637 rules, its effective type coverage is lower because the rules are defined with very fine granularity. For example, each sensitive permission check is treated as a separate rule. In addition, some vulnerability types are not covered by RULEDROID because they are not included in the official Android documentation or OWASP MASTG. For instance, the "Temp File Data Exposure" issue is excluded because it is a general problem, not a defect of the Android system itself or the Java language.

Performance on Benchmarks. We conducted a comparative analysis of RULEDROID and seven mainstream SAST tools across three benchmarks (Ghera, MASTG&PIVAA, and DroidCVE++) to comprehensively evaluate detection capabilities.

The evaluation results in Table V indicate that traditional SAST tools underperform RULEDROID in multiple metrics. Notably, the TP counts for the SAST tools are significantly lower than those for RULEDROID$_{1\text{-}3}$. For example, on the Ghera benchmark, APKHunt detected only 28 vulnerability cases, whereas RULEDROID$_{1\text{-}3}$ detected 40, 38, and 38 cases, respectively. Moreover, RULEDROID$_{1\text{-}3}$ achieve F1-scores of 76.92%, 76.00%, and 74.51% on Ghera, compared to 52.83% for APKHunt. Similar trends are observed for the recall rates on the MASTG&PIVAA and DroidCVE++ benchmarks. Furthermore, the traditional tools perform significantly worse than RULEDROID in terms of FDR, clearly demonstrating that RULEDROID is the top-performing detection tool across all three benchmarks.

This performance advantage primarily stems from the limitations of manually crafted rules in traditional SAST tools. First, because expert rule development is labor-intensive, these tools tend to focus on high-risk vulnerabilities while overlooking issues with relatively minor impact. Second, omissions during manual collection of vulnerability information can leave certain security issues unaddressed. Additionally, many older tools have outdated rules that fail to capture newly emerging vulnerabilities. In contrast, RULEDROID generates rules based on comprehensive Android security documentation, thereby covering both new and legacy issues and detecting more vulnerability instances. Furthermore, by integrating LLM-FPF for false-positive filtering, RULEDROID eliminates context-dependent false positives, such as those arising from the use of weak hashes in non-security scenarios. As a result, RULEDROID achieves significantly lower FPR and FDR than other tools.

TABLE V: Performance comparison of RULEDROID and other SAST tools

Tool	Ghera (59 cases)									MASTG&PIVAA (52 cases)					DroidCVE++ (278 cases)				
	TP	FN	FP	TN	Prec.	Recall	FPR	F1	#Rep	TP	FN	Recall	FDR	#Rep	TP	FN	Recall	FDR	#Rep
RULEDROID$_1$	40	19	5	54	88.89%	67.80%	8.47%	**76.92%**	689	**42**	**10**	**80.76%**	28.00%	203	**251**	**27**	**90.29%**	**16.00%**	14312
RULEDROID$_2$	38	21	**3**	**56**	**92.68%**	64.41%	**5.08%**	76.00%	613	40	12	76.92%	34.00%	183	245	33	88.13%	22.00%	13874
RULEDROID$_3$	38	21	5	54	88.37%	64.41%	8.47%	74.51%	789	39	13	75.00%	**26.00%**	217	246	32	88.49%	22.00%	15078
SPECK	16	43	6	53	72.73%	27.12%	10.17%	39.51%	54	18	34	34.62%	88.37%	61	102	176	36.69%	82.00%	22145
Marvin	18	41	5	54	78.26%	30.51%	8.47%	43.90%	203	14	38	26.92%	66.67%	13	126	152	45.32%	68.00%	17807
AndroBugs	15	44	4	55	78.95%	25.42%	6.78%	38.46%	379	16	36	30.77%	92.68%	57	162	116	58.27%	76.00%	42317
QARK	13	46	4	55	76.47%	22.03%	6.78%	34.21%	48705	10	42	19.23%	82.00%	2844	145	133	52.16%	76.00%	408103
MobSF	18	41	7	52	72.00%	30.51%	11.86%	42.86%	904	19	33	32.75%	86.00%	97	132	146	47.48%	66.00%	79139
AUSERA	21	38	7	52	75.00%	35.59%	11.86%	48.28%	543	22	30	42.30%	84.85%	55	186	92	66.91%	62.00%	61625
APKHunt	28	31	19	40	59.57%	47.46%	32.20%	52.83%	21995	25	27	48.08%	78.00%	1397	239	39	85.91%	60.00%	393933

Prec. = Precision; FPR = False Positive Rate; F1 = F1-score; FDR = False Discovery Rate; #Rep=Total Number of Reports.
Underlined values indicate the best performance among the compared results.

TABLE VI: Vulnerability type coverage across different tools.

Tool	Total	VulsTotal		MASWE	
		Coverage	Extra Rules	Coverage	Extra Rules
RULEDROID$_1$	406	**63/67**	289	**72/116**	280
RULEDROID$_2$	398	59/67	288	70/116	279
RULEDROID$_3$	415	61/67	292	70/116	280
SPECK	31	23/67	5	30/116	1
Marvin	64	28/67	22	29/116	19
AndroBugs	67	27/67	35	26/116	37
QARK	36	21/67	5	28/116	3
MobSF	637	39/67	544	46/116	525
AUSERA	50	40/67	10	43/116	8
APKHunt	74	45/67	34	45/116	32

Underlined values indicate the best performance among the compared results.

Short Answer to RQ3

Compared to traditional static detection techniques, RULEDROID demonstrates superior overall performance in Android security detection tasks. In particular, with respect to the number of detected vulnerability cases, RULEDROID exhibits significant advantages.

E. RQ4: Comparison with LLM-based Detections

This section compares the vulnerability detection performance of RULEDROID with other LLM-based Android app security detection approaches.

Evaluation Methodology. Since there are no other readily available LLM-based tools for Android app vulnerability detection, we adopted the detection method proposed by Kouliaridis et al. [42] as the baseline for comparison. The main approach enhances LLM detection performance through vulnerability-type knowledge and prompt engineering. We refer to this method as *LLM-Detector*. Since the detection capabilities of LLM-Detectors depend on their pre-training data, our evaluation focuses on detection effectiveness rather than the diversity of vulnerability types.

We conducted experiments on three benchmark datasets to compare RULEDROID with four LLM-Detectors built on advanced models (GPT-5, o4-mini, DeepSeek-V3.2, and DeepSeek-R1), referred to as LLM-Detector-GPT5, LLM-Detector-o4mini, LLM-Detector-V3.2, and LLM-Detector-R1, respectively. The evaluation aims to assess and compare the vulnerability detection performance of RULEDROID against these LLM-based baselines.

Performance Analysis. Overall, RULEDROID outperforms all evaluated LLM-based approaches. As shown in Table VIII, while LLMs leverage robust pattern recognition to identify many true vulnerabilities, their reliance on statistical weighting rather than in-depth code analysis often leads to a high false-positive rate, lowering the overall F1 score. For example, although DeepSeek-R1 reports 45 true positives, it also produces 32 false positives, resulting in an F1 score of only 66.18%, whereas RULEDROID$_{1\text{-}3}$ consistently achieve F1 scores above 70%. Moreover, on the MASTG&PIVAA and DroidCVE++ benchmarks, RULEDROID$_{1\text{-}3}$ attain the highest recall rates and lowest FDR, clearly demonstrating the superior performance of RULEDROID in practical detection scenarios.

We observed that many false positives produced by LLM-Detectors stem from incorrect judgment of vulnerability types. For example, in GPT-5's results on the DroidCVE++ benchmark, 85 out of 97 false negatives (87.62%) were due to incorrect type judgments. This reflects a key limitation of LLM-based detection: these models often rely on matching features, such as suspicious operations or sensitive functions, learned from training data. When the input code is long and exhibits characteristics of multiple vulnerability types, the model is more likely to misclassify them.

This limitation also explains why the performance of LLM-Detectors on the Ghera and MASTG&PIVAA benchmarks is only about 15% lower than that of RULEDROID, while a much larger gap appears on DroidCVE++. Even LLM-Detector-GPT5 achieves only 65.11% recall, significantly lower than the 90.29%, 88.13%, and 88.49% recall of RULEDROID$_{1\text{–}3}$. One contributing factor is that real-world code files in DroidCVE++ are often large, sometimes reaching the model's token limit, which causes certain vulnerabilities to be ignored. In addition, when long code snippets contain features resembling multiple vulnerability types, the model may struggle to pinpoint the actual issue.

TABLE VII: Vulnerability types supported by each tool.

Vulnerability Type	RuleDroid$_1$	RuleDroid$_2$	RuleDroid$_3$	SPECK	MobSF	Marvin	AndroBugs	APKHunt	AUSERA	QARK
WebView Password Exposure	⋆	⋆	⋆			⋆			⋆	
Logging Data Exposure	⋆	⋆	⋆		⋆			⋆	⋆	
External/Internal Data Exposure	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	
Cache Data Disclosure	⋆	⋆	⋆	⋆				⋆		
Temp File Data Exposure					⋆			⋆	⋆	
SQLite Data Exposure	⋆	⋆	⋆					⋆	⋆	
SMS Data Exposure	⋆	⋆	⋆	⋆	⋆		⋆		⋆	
Clipboard Data Exposure	⋆	⋆	⋆		⋆			⋆		
Hardcoded IP Exposure	⋆	⋆	⋆		⋆			⋆		
Hardcoded Email Exposure	⋆	⋆	⋆					⋆		
Device ID Exposure	⋆	⋆	⋆				⋆			
Android ID Exposure	⋆	⋆	⋆				⋆			
Hardcoded URL Exposure	⋆	⋆	⋆					⋆		
Hardcoded Sensitive Data Exposure	⋆	⋆	⋆		⋆	⋆		⋆		
Insecure Base64 Encryption	⋆		⋆				⋆		⋆	
Insecure Blowfish Encryption	⋆				⋆				⋆	
Improper Handle DES Encryption	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆
Improper Handle AES Encryption	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆
Improper Handle RSA Encryption	⋆	⋆	⋆		⋆				⋆	
Improper Handle RC4 Encryption	⋆	⋆	⋆		⋆	⋆		⋆		
Improper Handle Insecure Hash	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	
Use Insecure Random	⋆	⋆	⋆	⋆	⋆	⋆		⋆	⋆	
Weak CBC Cipher Modes	⋆	⋆	⋆	⋆	⋆			⋆		
Hardcoded IV Issue	⋆	⋆	⋆		⋆	⋆		⋆		
Improper Package Hardcoded	⋆	⋆	⋆		⋆		⋆	⋆	⋆	⋆
Misuse Empty Pending Intent Issue	⋆	⋆	⋆						⋆	⋆
Improper Receiver Registration	⋆	⋆	⋆			⋆			⋆	
Misuse Implicit Intent Issue	⋆	⋆	⋆			⋆		⋆	⋆	
Exported Not Protected Components	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
Unprotected Content Provider	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
Sticky Broadcast Intent Issue	⋆	⋆	⋆	⋆		⋆				⋆
ContentProvider Permissions Issue	⋆	⋆	⋆		⋆					⋆
Manifest Screenshot Harvest	⋆		⋆		⋆		⋆	⋆	⋆	
Manifest Backup Issue	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆
Manifest Debug Issue	⋆	⋆	⋆		⋆	⋆	⋆	⋆	⋆	⋆
Mode World Storage Readable Issue	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
Mode World Storage Writable Issue	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
Dynamic Code Loading Issue	⋆	⋆	⋆	⋆		⋆	⋆		⋆	
Runtime Command Execution Issue	⋆	⋆	⋆			⋆	⋆	⋆	⋆	
Rooted Device Detection					⋆		⋆	⋆		
Super User Privileges	⋆	⋆	⋆		⋆					
Sensitive Functionality (loadlibrary)	⋆						⋆		⋆	
SQL Injection	⋆	⋆	⋆	⋆	⋆			⋆	⋆	
Fragment Injection	⋆	⋆	⋆			⋆	⋆	⋆	⋆	
ContentProvider Openfile	⋆	⋆	⋆			⋆			⋆	
Using HTTP Issue	⋆	⋆	⋆	⋆			⋆	⋆	⋆	
ClearText Traffic Issue	⋆	⋆	⋆	⋆	⋆			⋆		
Debug CA Configuration Issue	⋆	⋆	⋆	⋆	⋆					
Use Expired Certificate		⋆	⋆						⋆	
Use SHA1_MD5 Certificate	⋆	⋆	⋆		⋆				⋆	
Android Debug Certificate	⋆	⋆	⋆		⋆					
Insecure AllowUserCA	⋆	⋆	⋆		⋆			⋆		
Use Insecure Socket	⋆	⋆	⋆	⋆					⋆	
Use Firebase exposed	⋆	⋆	⋆		⋆			⋆		
Use Insecure SSL Socket Factory	⋆	⋆	⋆			⋆	⋆	⋆		
Use Invalid Hostname Verification	⋆	⋆	⋆			⋆	⋆	⋆	⋆	⋆
Use Invalid Server Verification	⋆	⋆	⋆			⋆	⋆	⋆	⋆	⋆
Use Allow All Hostname Verification	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆
WebView Cert Validation Issue	⋆	⋆	⋆		⋆	⋆	⋆	⋆		
WebView SOP Warning	⋆									⋆
WebView JavaScript Execution	⋆	⋆	⋆	⋆	⋆		⋆	⋆	⋆	⋆
WebView Java Objects Exposure	⋆	⋆	⋆	⋆	⋆	⋆	⋆	⋆		⋆
WebView Insecure Load Plugin									⋆	⋆
WebView Local File Access	⋆	⋆	⋆			⋆	⋆	⋆	⋆	⋆
WebView Local File Cleanup	⋆	⋆	⋆	⋆				⋆		
WebView Insecure URL Loading	⋆	⋆	⋆					⋆		⋆
WebView Remote Debugging	⋆	⋆	⋆		⋆			⋆		

TABLE VIII: Performance comparison of RULEDROID and LLM-based tools.

Tool	Ghera (59 cases)									MASTG&PIVAA (52 cases)					DroidCVE++ (278 cases)				
	TP	FN	FP	TN	Prec.	Recall	FPR	F1	#Rep	TP	FN	Recall	FDR	#Rep	TP	FN	Recall	FDR	#Rep
RULEDROID$_1$	40	19	5	54	88.89%	67.80%	8.47%	**76.92%**	689	**42**	**10**	**80.76%**	18.00%	203	**251**	**27**	**90.29%**	**10.00%**	14312
RULEDROID$_2$	38	21	**3**	**56**	**92.68%**	64.41%	**5.08%**	76.00%	613	40	12	76.92%	24.00%	183	245	33	88.13%	16.00%	13874
RULEDROID$_3$	38	21	5	54	88.37%	64.41%	8.47%	74.51%	789	39	13	75.00%	16.00%	217	246	32	88.49%	16.00%	15078
LLM-Detector-GPT5	42	17	26	33	61.76%	71.19%	44.07%	66.14%	338	33	19	63.46%	68.75%	65	181	97	65.11%	62.00%	26,640
LLM-Detector-o4mini	32	27	16	43	66.67%	54.24%	27.12%	59.81%	158	32	20	61.54%	71.43%	54	180	98	64.75%	64.00%	16,657
LLM-Detector-V3.2	40	19	25	34	61.54%	67.80%	42.37%	64.52%	280	36	16	69.63%	70.00%	56	175	103	62.95%	72.00%	22,791
LLM-Detector-R1	**45**	**14**	32	27	58.44%	**76.27%**	54.24%	66.18%	316	37	15	71.15%	57.89%	56	173	105	62.23%	74.00%	26715

Prec. = Precision; FPR = False Positive Rate; F1 = F1-score; FDR = False Discovery Rate; #Rep=Total Number of Reports.
Underlined values indicate the best performance among the compared results.

TABLE IX: Comparison of overhead between RULEDROID and LLM-based tools.

Tool	Token (Millions)		Time
	Input	Output	
RULEDROID	14.69	2.80	36 minutes
LLM-Detector-GPT5	943.63	9.93	303 minutes
LLM-Detector-o4mini	943.63	6.46	259 minutes
LLM-Detector-V3.2	943.63	10.87	138 minutes
LLM-Detector-R1	943.63	13.15	409 minutes

For example, in `CVE-2020-35454` from DroidCVE++, the AndroidManifest file exceeds 400 lines and contains several suspicious patterns. None of the four LLM-Detectors identified the real issue caused by `android:allowBackup="true"`. Instead, all focused on `android:exported="true"`, which seemed risky but was not the root cause, leading to false positives.

Overhead Analysis. In addition to detection effectiveness, we evaluated the total time and token consumption of RULEDROID and other LLM-based methods across the three benchmarks in identical environments using 60 concurrent threads. As shown in Table IX, the results for RULEDROID (averaged across RULEDROID$_{1-3}$) demonstrate significantly lower token overhead than other LLM tools. This efficiency arises because during the scanning phase, RULEDROID invokes the LLM-FPF only to filter out false-positive reports. Consequently, RULEDROID consumes only 14.69M input tokens and 2.80M output tokens. In contrast, direct LLM approaches require substantially more token usage: all LLM detectors consume 943.63M input tokens, with output tokens ranging from 6.46M to 13.15M. Furthermore, RULEDROID exhibits superior time efficiency, requiring significantly less execution time than the direct LLM-based methods.

Short Answer to RQ4

Compared with direct vulnerability detection by the tested LLMs, RULEDROID provides better results on all three benchmarks. Its most significant advantage shows up in real-world code, where LLMs face token limits and may overlook important details. By using clearly defined static rules and knowledge bases, RULEDROID avoids these issues and offers more reliable detection overall.

F. RQ5: Discovery of Real-world Vulnerabilities

To evaluate RULEDROID's practical detection capability, we randomly selected 100,000 Android apps from the AndroZoo dataset [43]. Using RULEDROID$_1$, we performed a comprehensive scan. From the 183,781 `ERROR`-level reports, we randomly sampled 500 issues for manual validation. Statistically, this sample size provides a 95% confidence level with a margin of error of approximately 4.38%. The manual validation was conducted independently by two security experts. A reported issue was classified as a TP only if both experts verified the vulnerability's existence. Through this rigorous process, we identified 417 true vulnerabilities and 83 false positives.

To empirically compare RULEDROID against existing methods in real-world apps, we used these 500 validated reports as a localized ground-truth dataset comprising 417 positive and 83 negative instances. We executed the baseline static analysis tools and direct LLM detectors on the corresponding apps to evaluate their performance on this specific subset. The comparative results are summarized in Table X. The evaluation indicates that the other tools missed a significant portion of the vulnerabilities identified by RULEDROID.

Among the 417 true cases, 62 exploitable issues could cause real-world impact and were responsibly disclosed to the affected vendors and the CVE Program (MITRE). To date, 57 CVE IDs have been assigned, spanning 10 distinct vulnerability types. Our empirical comparison confirms that these critical issues are largely undetectable by the existing static analysis tools we evaluated. For example, `CVE-2025-7889` involves a task-hijacking vulnerability caused by a misconfiguration in the Android Manifest that none of the other tools detected.

Short Answer to RQ5

In practical vulnerability detection scenarios, RULEDROID performs strongly, discovering several new vulnerabilities that other tools miss.

VI. DISCUSSION AND LIMITATIONS

Although RULEDROID demonstrates significant progress over similar tools regarding rule coverage, maintainability, and detection capabilities, it still faces some limitations. Addressing them will be the focus of future research and optimization.

TABLE X: Evaluation on real-world apps.

Tool	TP	FP	Precision	F1-score
RULEDROID$_1$	417	83	83.40%	90.95%
SPECK	130	39	76.92%	44.37%
Marvin	175	38	82.16%	55.56%
AndroBugs	191	41	82.33%	58.86%
QARK	194	39	83.26%	59.69%
MobSF	207	43	82.80%	62.07%
AUSERA	237	41	85.25%	68.20%
APKHunt	262	65	80.12%	70.43%
LLM-Detector-GPT5	279	69	80.17%	72.94%
LLM-Detector-o4mini	262	52	83.44%	71.68%
LLM-Detector-V3.2	261	72	78.38%	69.60%
LLM-Detector-R1	255	66	79.44%	69.11%

Limitations of Static Rule-based Detection. As RULEDROID ultimately relies on static rules for vulnerability scanning, it inherently suffers from the limitations of this approach. In particular, static analysis provides limited insight into runtime behaviors and dynamic context, such as the use of Java reflection, which can obscure the actual control and data flows from static analyzers. Furthermore, Semgrep's pattern language fundamentally constrains the expressiveness of the generated detection rules when tracking complex execution paths. Consequently, RULEDROID struggles to detect vulnerability classes that demand deep semantic understanding or cross-context tracking. Prominent examples that fall fundamentally out of scope include vulnerabilities hidden within intricate asynchronous callback chains, cross-language data flows via the JNI, and complex ICC flows. These subtle logical flaws, which manifest primarily during dynamic program execution, remain highly challenging to capture using static pattern matching.

Code Protection Techniques. When apps employ code protection techniques such as code obfuscation and packing, the app's original semantic structure, class names, and variable identifiers are significantly altered. This consequently degrades the effectiveness of our detection rules, which rely on API invocation patterns and semantic features. Particularly for packed apps, static analysis is rendered entirely ineffective without a preceding unpacking process.

Detecting Unknown Vulnerabilities. RULEDROID is designed to generate detection rules from official Android security documents and widely accepted guidelines. Its goal is to cover known vulnerability categories, not to discover entirely new types. However, RULEDROID remains responsive to new threats: as soon as a new vulnerability pattern is documented, RULEDROID can generate corresponding rules.

Limitations of Rule Evolution. Because the update mechanism in RULEDROID is document-driven, the system does not function as a general rule-evolution framework. The tool cannot automatically adapt existing rules, infer version-specific changes in API behavior, resolve semantic conflicts, or deprecate previously valid rules unless these updates are explicitly documented in the source documents. Consequently, the update capabilities of RULEDROID depend on the completeness and timeliness of the underlying documentation.

Further Improvements in Rule Quality. Although RULEDROID outperforms various static analysis tools overall, and the vast majority of its generated rules are semantically correct, the automatically generated rules are still less refined than those carefully crafted by human experts. In practical deployments, incorporating expert knowledge to refine these rules can enhance precision and reduce false positives. Crucially, providing experts with these generated rules as a baseline substantially reduces the manual effort required compared to deriving rules from documentation from scratch. However, our goal was to demonstrate the effectiveness of fully automated rule generation. Therefore, *no manual refinements were applied in our experiments*. Future work may explore a hybrid approach that leverages both automated processes and targeted expert input to achieve higher detection quality.

Furthermore, since the distinct rule sets generated by RULEDROID contain numerous distinct and orthogonal rules, computing the intersection and union of these rules, or of their detection reports, via a dedicated mechanism offers a potential avenue for improving both rule quality and detection efficacy. However, for this approach to be viable, it remains necessary to develop specific methods capable of computing the semantic equivalence of the detection rules or vulnerability reports.

VII. RELATED WORK

Static Analysis for Android Apps. Static analysis techniques have long formed the foundation for detecting Android vulnerabilities, with early and recent work aiming to improve accuracy, coverage, and real-world applicability. For example, Lee et al. [44] investigated inter-app communication flaws through static analysis and visualization, while Lenk et al. [45] focused on misconfigured content providers using static analysis combined with attack validation to expose privacy leaks. Yang et al. [46] applied an XML schema and domain-aware NLP rules to detect Android Manifest misconfigurations. Wang et al. [47] utilized static analysis techniques to analyze Android native code. Wu et al. [48] developed a static analysis tool Relda2 to detect resource leaks in Android. Liu et al. [49] proposed ArchiDroid, utilizing graph convolution networks to predict missing activity transitions and improve static GUI model completeness. Yan et al. [50] presented CrashTracker, which locates framework-specific crashes via static analysis of exception summaries and crucial variables. Chen [51] analyzed privacy leakage in Android logs to provide empirical foundations for expanding static detection rules.

Efforts to evaluate and enhance SAST tools have led to contributions such as the VulsTotal platform by Zhu et al. [36], which unifies vulnerability categories and tool outputs and introduces a CVE-based benchmark for systematic evaluation. AUSERA [3] developed a vulnerability taxonomy and enhanced detection semantics to reduce false positives and improve accuracy across 50 vulnerability types. Samhi et al. [52] investigated the soundness of call graph construction across various Android static analysis tools, highlighting significant limitations in capturing dynamic execution methods.

Attention has also been given to addressing vulnerabilities in third-party libraries. ATVHunter [53] used a two-phase

approach based on control-flow graphs and opcode features, and MtdScout [54] employed method-level clone detection to identify weaknesses that traditional analyses might miss. Furthermore, Draschbacher et al. [55] examined the Code Transparency mechanism in the AAB distribution model, revealing signature verification flaws that could permit stealthy code injection.

In contrast to previous SAST tools relying on manually defined detection rules, RULEDROID integrates RAG and workflow reasoning to automatically generate and continuously adapt its rule set as Android evolves. This overcomes the limitations of traditional tools that struggle to keep pace with new vulnerabilities.

LLM-driven Vulnerability Detection. LLMs have shown strong code understanding and generation capabilities, prompting their application in vulnerability detection. Several studies have evaluated LLM performance on general detection tasks; for instance, work by Guo et al. [56] and Lin et al. [57] compared models of various scales, architectures, and fine-tuning strategies across multiple programming languages. Their findings suggest that smaller LLMs may outperform larger ones in specific scenarios, and overall performance is sensitive to data quality and context window size. Yang et al. [58] presented KNighter, which utilizes LLMs to synthesize static analysis checkers from historical patches, solving scalability issues via a multi-stage validation mechanism. Li et al. [59] developed MoCQ, a neuro-symbolic framework combining LLMs with tools like Joern and CodeQL, which successfully identified real-world vulnerabilities and patterns missed by human experts.

Other approaches have developed detection frameworks for specific vulnerabilities. CryptoLLM [60] combines static code slicing with CodeT5 to detect cryptographic API misuse, outperforming traditional rule-based tools on both benchmark datasets and real Android apps. GPTScan [5] leverages GPT to uncover logical vulnerabilities in smart contracts and uses static analysis to cut false positives, identifying issues that human audits had overlooked. AdvSCanner [61] integrates LLMs with static analysis to generate adversarial smart contracts (ASCs) for detecting reentrancy flaws, achieving an 85.9% success rate and reducing audit times. DoHunter [7] combines LLM-based context with expert-designed features to detect encrypted DNS tunnel traffic associated with advanced persistent threats. Asmita et al. [62] have incorporated LLMs into fuzzing by generating target-specific seeds and reusing crash data, thus boosting vulnerability discovery in systems like BusyBox. Liu et al. [9] introduce a new framework called GPTAid, which aims to automatically generate API Parameter Security Rules (APSRs) through LLM and detect API abuse issues caused by improper parameter usage.

In contrast to direct LLM-based detection approaches, RULEDROID uses LLMs only to generate high-quality rules rather than directly identifying vulnerabilities. By performing the actual detection through rigorous static analysis, it ensures higher reliability.

VIII. CONCLUSION

Securing Android apps remains challenging due to the platform's expansive API surfaces and diverse threats. In this paper, we introduced RULEDROID, an approach that integrates static analysis with LLM-generated detection rules to address Android security vulnerabilities. RULEDROID automatically adapts to evolving security requirements without manual intervention, and evaluations across three benchmarks demonstrate its superior performance over traditional SAST tools and direct LLM-based detection approaches. In real-world testing, RULEDROID uncovered numerous vulnerabilities. We responsibly reported 62 cases and have already received 57 CVE ID assignments. Additionally, we present DroidCVE++, an enhanced benchmark that expands the types of vulnerabilities and provides verifiable location details, offering a more realistic evaluation framework for the security community.

REFERENCES

[1] (2023) MobSF. Accessed: 2025-02-18. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[2] (2023) APKHunt. Accessed: 2025-03-25. [Online]. Available: https://github.com/Cyber-Buddy/APKHunt

[3] S. Chen, Y. Zhang, L. Fan, J. Li, and Y. Liu, "AUSERA: Automated Security Risk Assessment For Vulnerability Detection In Android Apps," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE), Rochester, MI USA, 2022*, 2022, pp. 1–5.

[4] C. Fang, N. Miao, S. Srivastav, J. Liu, R. Zhang, R. Fang, Asmita, R. Tsang, N. Nazari, H. Wang, and H. Homayoun, "Large Language Models for Code Analysis: Do LLMs Really Do Their Job?" in *Proceedings of the 33rd USENIX Security Symposium (USENIX-SEC), Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[5] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "GPTScan: Detecting Logic Vulnerabilities In Smart Contracts By Combining GPT With Program Analysis," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE), Lisbon, Portugal, April 14–20, 2024*, 2024, pp. 1–13.

[6] A. Stafeev, T. Recktenwald, G. D. Stefano, S. Khodayari, and G. Pellegrino, "YuraScanner: Leveraging LLMs for Task-driven Web App Scanning," in *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-28, 2025*, 2025.

[7] J. Diao, S. Zhao, J. Xie, R. Xie, and G. Shi, "Poster: DoHunter: A Feature Fusion-Based LLM For DoH Tunnel Detection," in *Proceedings of the 31st ACM SIGSAC Conference on Computer and Communications Security (CCS), Salt Lake City, UT, USA, October 14–18, 2024*, 2024, pp. 5012–5014.

[8] Y. Yang, J. Liu, K. Chen, and M. Lin, "The Midas Touch: Triggering the Capability of LLMs for RM-API Misuse Detection," in *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-28, 2025*, 2025.

[9] J. Liu, Y. Yang, K. Chen, and M. Lin, "Generating API Parameter Security Rules with LLM for API Misuse Detection," in *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-28, 2025*, 2025.

[10] A. Mechri, M. A. Ferrag, and M. Debbah, "SecureQwen: Leveraging LLMs for vulnerability detection in python codebases," *Computers & Security*, vol. 148, p. 104151, 2025.

[11] S. Ullah, M. Han, S. Pujar, H. Pearce, A. K. Coskun, and G. Stringhini, "LLMs Cannot Reliably Identify And Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, And Benchmarks," in *Proceedings of the 45th IEEE Symposium on Security and Privacy (IEEE S&P), San Francisco, CA, USA, May 19–23, 2024*, 2024, pp. 862–880.

[12] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung, "Survey of Hallucination in Natural Language Generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 248:1–248:38, 2023.

[13] G. Brauwers and F. Frasincar, "A General Survey on Attention Mechanisms in Deep Learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 3279–3298, 2023.

[14] (2025) Android Developer Documentation. Accessed: 2025-03-04. [Online]. Available: https://developer.android.com/

[15] M. Grunde-McLaughlin, M. S. Lam, R. Krishna, D. S. Weld, and J. Heer, "Designing LLM chains by adapting techniques from crowdsourcing workflows," *ACM Trans. Comput. Hum. Interact.*, vol. 32, pp. 27:1–27:57, 2025.

[16] P. S. H. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Proceedings of the Thirty-Fourth Annual Conference on Neural Information Processing Systems (NeurIPS), virtual, December 6-12, 2020*, 2020.

[17] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval Augmentation Reduces Hallucination in Conversation," in *Proceedings of The 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP), Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, 2021, pp. 3784–3803.

[18] R. Rossini, S. Pizzi, S. Doria, M. Conti, and E. Losiouk, "Poster: SPECK: From Google Textual Guidelines to Automatic Detection of Android Apps Vulnerabilities," in *Proceedings of the 22nd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Graz, Austria, July 9-11, 2025*, 2025, pp. 167–172.

[19] (2016) Marvin-Static-Analyzer. Accessed: 2025-03-25. [Online]. Available: https://github.com/programa-stic/Marvin-static-Analyzer

[20] (2025) StrandHogg Task Hijacking Vulnerability. Accessed: 2025-03-03. [Online]. Available: https://developer.android.com/privacy-and-security/risks/strandhogg

[21] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving API Caveats Accessibility By Mining API Caveats Knowledge Graph," in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, September 23-29, 2018*, 2018, pp. 183–193.

[22] (2025) Semgrep. Accessed: 2025-04-08. [Online]. Available: https://semgrep.dev/

[23] (2025) CodeQL: Semantic Code Analysis. Accessed: 2025-04-08. [Online]. Available: https://codeql.github.com/

[24] L. Richardson. (2025) Beautiful Soup Documentation. Accessed: 2025-03-04. [Online]. Available: https://beautiful-soup-4.readthedocs.io/en/latest/

[25] (2025) OWASP Mobile Application Security Testing Guide (MASTG). Accessed: 2025-02-23. [Online]. Available: https://github.com/OWASP/owasp-mastg/blob/master/Document/

[26] (2025) Explore | Semgrep. Accessed: 2025-04-08. [Online]. Available: https://semgrep.dev/explore

[27] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[28] OpenAI, "text-embedding-3-large," 2024, accessed: 2025-08-02. [Online]. Available: https://platform.openai.com/docs/models/text-embedding-3-large

[29] (2025) semgrep-rules-android-security. Accessed: 2025-04-08. [Online]. Available: https://github.com/mindedsecurity/semgrep-rules-android-security

[30] (2025) tomd: Markdown renderer for HTML/XML. Accessed: 2025-04-08. [Online]. Available: https://github.com/elliotgao2/tomd

[31] (2017) Jadx. Accessed: 2025-03-25. [Online]. Available: https://github.com/skylot/jadx

[32] (2014) RFC 7232 - Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. Accessed: 2025-03-15. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7232

[33] (2022) MASTG Hacking Playground. Accessed: 2025-03-15. [Online]. Available: https://github.com/OWASP/MASTG-Hacking-Playground

[34] (2023) PIVAA. Accessed: 2025-03-03. [Online]. Available: https://github.com/HTBridge/pivaa

[35] J. Mitra and V. Ranganath, "Ghera: A Repository of Android App Vulnerability Benchmarks," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), Toronto, Canada, November 8, 2017*, 2017, pp. 43–52.

[36] J. Zhu, K. Li, S. Chen, L. Fan, J. Wang, and X. Xie, "A Comprehensive Study On Static Application Security Testing (SAST) Tools For Android," *IEEE Transactions on Software Engineering*, vol. 50, no. 12, pp. 3385–3402, 2024.

[37] "MASTG-BEST-0008: Debugging Disabled for WebViews," 2026, accessed: 2026-05-11. [Online]. Available: https://mas.owasp.org/MASTG/best-practices/MASTG-BEST-0008

[38] (2015) AndroBugs. Accessed: 2025-03-25. [Online]. Available: https://github.com/AndroBugs/AndroBugs_Framework

[39] (2018) QARK. Accessed: 2025-04-08. [Online]. Available: https://github.com/linkedin/qark/

[40] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 259–269.

[41] (2024) OWASP Mobile Application Security Weakness Enumeration (MASWE). Accessed: 2025-07-19. [Online]. Available: https://github.com/OWASP/maswe

[42] V. Kouliaridis, G. Karopoulos, and G. Kambourakis, "Assessing the Effectiveness of LLMs in Android Application Vulnerability Analysis," in *Proceedings of the 7th International Conference on Attacks and Defenses for the Internet-of-Things (ADIoT), Hangzhou, China, December 13–14, 2024*, 2024, pp. 139–154.

[43] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting Millions Of Android Apps For The Research Community," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR), Austin, Texas, May 14–15, 2016*, 2016, pp. 468–471.

[44] Y. K. Lee, P. Yoodee, A. Shahbazian, D. Nam, and N. Medvidovic, "SEALANT: A Detection And Visualization Tool For Inter-App Security Vulnerabilities In Android," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, October 30–November 3, 2017*, 2017, pp. 883–888.

[45] C. Lenk and J. Kinder, "Poster: Privacy Risks From Misconfigured Android Content Providers," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS), Copenhagen, Denmark, November 26–30, 2023*, 2023, pp. 3579–3581.

[46] Y. Yang, M. Elsabagh, C. Zuo, R. Johnson, A. Stavrou, and Z. Lin, "Detecting and Measuring Misconfigured Manifests In Android Apps," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, November 7–11, 2022*, 2022, pp. 3063–3077.

[47] J. Wang and H. Wang, "NativeSummary: Summarizing Native Binary Code for Inter-language Static Analysis of Android Apps," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, September 16-20, 2024*, 2024, pp. 971–982.

[48] T. Wu, J. Liu, X. Deng, J. Yan, and J. Zhang, "Relda2: An Effective Static Analysis Tool for Resource Leak Detection in Android Apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, September 3-7, 2016*, 2016, pp. 762–767.

[49] Z. Liu, C. Chen, J. Wang, Y. Su, Y. Huang, J. Hu, and Q. Wang, "Ex pede herculem: Augmenting activity transition graph for apps via graph convolution network," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), Melbourne, Australia, May 14-20, 2023*, 2023, pp. 1983–1995.

[50] J. Yan, M. Wang, Y. Liu, J. Yan, and L. Zhang, "Locating framework-specific crashing faults with compact and explainable candidate set," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), Melbourne, Australia, May 14-20, 2023*, 2023, pp. 172–183.

[51] Z. Chen, B. Ray, and M. Zhou, "A comprehensive study of privacy leakage vulnerability in android app logs," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), Sacramento, CA, USA, October 27 - November 1, 2024*, 2024, pp. 2510–2513.

[52] J. Samhi, R. Just, T. F. Bissyandé, M. D. Ernst, and J. Klein, "Call graph soundness in android static analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Vienna, Austria, September 16-20, 2024*, 2024, pp. 945–957.

[53] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "ATVHUNTER: Reliable Version Detection Of Third-Party Libraries For Vulnerability Identification In Android Applications," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE), Madrid, Spain, May 22–30, 2021*, 2021, pp. 1695–1707.

[54] Z. Zhang, H. Ma, D. Wu, D. Gao, X. Yi, Y. Chen, Y. Wu, and L. Jiang, "MtdScout: Complementing The Identification Of Insecure Methods In Android Apps Via Source-To-Bytecode Signature Generation And Tree-Based Layered Search," in *Proceedings of the 9th IEEE European*

Symposium on Security and Privacy (EuroS&P), Vienna, Austria, July 8–12, 2024, 2024, pp. 724–740.

[55] F. Draschbacher and L. Maar, "Manifest Problems: Analyzing Code Transparency For Android Application Bundles," in *Proceedings of the 40th Annual Computer Security Applications Conference (ACSAC), Honolulu, HI, USA, December 9–13, 2024*, 2024, pp. 1109–1122.

[56] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino, "Outside The Comfort Zone: Analysing LLM Capabilities In Software Vulnerability Detection," in *Proceedings of the 29th European Symposium on Research in Computer Security (ESORICS), Bydgoszcz, Poland, September 16–20, 2024*, 2024, pp. 271–289.

[57] J. Lin and D. Mohaisen, "From Large To Mammoth: A Comparative Evaluation Of Large Language Models In Vulnerability Detection," in *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24–28, 2025*, 2025.

[58] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang, "Knighter: Transforming static analysis with llm-synthesized checkers," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP), Lotte Hotel World, Seoul, Republic of Korea, October 13-16, 2025*, 2025, pp. 655–669.

[59] P. Li, S. Yao, J. S. Korich, C. Luo, J. Yu, Y. Cao, and J. Yang, "Automated static vulnerability detection via a holistic neuro-symbolic approach," *CoRR*, 2025.

[60] H. Baek, M. Lee, and H. Kim, "CryptoLLM: Harnessing the Power of LLMs to Detect Cryptographic API Misuse," in *Proceedings of the 29th European Symposium on Research in Computer Security (ESORICS), Bydgoszcz, Poland, September 16–20, 2024*, 2024, pp. 353–373.

[61] Y. Wu, X. Xie, C. Peng, D. Liu, H. Wu, M. Fan, T. Liu, and H. Wang, "AdvSCanner: Generating Adversarial Smart Contracts To Exploit Reentrancy Vulnerabilities Using LLM And Static Analysis," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), Sacramento, CA, USA, October 27–November 1, 2024*, 2024, pp. 1019–1031.

[62] Asmita, Y. Oliinyk, M. Scott, R. Tsang, C. Fang, and H. Homayoun, "Fuzzing BusyBox: Leveraging LLM And Crash Reuse For Embedded Bug Unearthing," in *Proceedings of the 33rd USENIX Security Symposium (USENIX-SEC), Philadelphia, PA, USA, August 14–16, 2024*, 2024.

Zhentao Xie received the B.E. degree in Network Engineering from South China Normal University in 2023. He is currently working toward a Master's degree in the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interests include mobile security and LLM-driven security analysis.

Mingyang Chen received the B.E. degree in Cyber Science and Technology from Qufu Normal University in 2024. He is currently working toward a Master's degree in the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interests include mobile security and LLM-driven code analysis.

Yaqi Gao is currently an undergraduate student in the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interests include mobile security.

Shishuai Yang received the Ph.D. degree in Cyber Science and Technology from Shandong University in 2025. He is currently a Lecturer in the School of Computer Science, Zhengzhou University of Aeronautics, Zhengzhou, China. His research interests include mobile security and IoT security.

Wenrui Diao received the Ph.D. degree in Information Engineering from The Chinese University of Hong Kong in 2017. He is currently a Professor in the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research interests include mobile security and IoT security. He received the 2019 ACM SIGSAC China Rising Star Award and the ACM CCS 2024 Distinguished Paper Award.

Xiangyu Liu received the Ph.D. degree in Information Engineering from The Chinese University of Hong Kong in 2016. He is currently a Senior Algorithm Expert in the Security Department of Alibaba Group, Hangzhou, China. His research interests include cybersecurity, artificial intelligence, and deep learning.

Kehuan Zhang received the Ph.D. degree in Informatics from Indiana University Bloomington in 2012. He is currently a Professor in the Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong. His research focuses on software and system security, including binary code analysis, mobile computing security, and IoT security.

APPENDIX

A. CVE List

TABLE X: CVE List

CVE ID	Package Name	Vulnerability Type
CVE-2025-5500	com.xiaoxun.xunoversea.mibrofit	StrandHogg Attack
CVE-2025-7889	caller.id.phone.number.block	StrandHogg Attack
CVE-2025-7890	com.dunamu.stockplus	StrandHogg Attack
CVE-2025-7891	com.instantbits.cast.webvideo	StrandHogg Attack
CVE-2025-7892	de.idnow	StrandHogg Attack
CVE-2025-7893	pro.foresightnews.app	StrandHogg Attack
CVE-2025-7940	com.house.auscat	StrandHogg Attack
CVE-2025-8207	com.canarabank.mobility	StrandHogg Attack
CVE-2025-8210	com.yeelight.cherry	StrandHogg Attack
CVE-2025-8257	com.maverick.lobby	StrandHogg Attack
CVE-2025-8258	com.sdmagic.number	StrandHogg Attack
CVE-2025-8275	bsc.devy.peru_cocktails	StrandHogg Attack
CVE-2025-8512	hk.com.tvb.bigbigshop	StrandHogg Attack
CVE-2025-8513	com.caixin.news	StrandHogg Attack
CVE-2025-8523	com.fruitcrush.fun	StrandHogg Attack
CVE-2025-8524	com.boquanhash.dotwallet	StrandHogg Attack
CVE-2025-8707	com.huuge.game.zjbox	StrandHogg Attack
CVE-2025-8745	com.ricepo.app	StrandHogg Attack
CVE-2025-9093	com.buzzfeed.android	StrandHogg Attack
CVE-2025-9097	com.cic_prod.bad	StrandHogg Attack
CVE-2025-9098	com.elseplus.filerecovery	StrandHogg Attack
CVE-2025-9102	com.mail.mobile.android.mail	StrandHogg Attack
CVE-2025-9134	com.aftership.AfterShip	StrandHogg Attack
CVE-2025-9135	de.hafas.android.vvt	StrandHogg Attack
CVE-2025-9671	com.passport.cash	StrandHogg Attack
CVE-2025-9672	de.hafas.android.rejseplanen	StrandHogg Attack
CVE-2025-9673	com.kakao.i.connect	StrandHogg Attack
CVE-2025-9674	com.hatsune.eagleee	StrandHogg Attack
CVE-2025-9675	com.tuyangkeji.changevoice	StrandHogg Attack
CVE-2025-9676	com.ncsoft.universeapp	StrandHogg Attack
CVE-2025-9677	com.duige.hzw.multilingual	StrandHogg Attack
CVE-2025-9695	com.thinkyeah.galleryvault	StrandHogg Attack
CVE-2025-10195	com.seismic.doccenter	StrandHogg Attack
CVE-2025-10715	com.ape_edication	StrandHogg Attack
CVE-2025-10716	com.cxsw.sdprinter	StrandHogg Attack
CVE-2025-10717	com.intsig.camscanner	StrandHogg Attack
CVE-2025-10718	com.ooma.office2	StrandHogg Attack
CVE-2025-10721	org.dayup.stocks	StrandHogg Attack
CVE-2025-10722	com.dw.android.mukbee	StrandHogg Attack
CVE-2026-4216	air.SmartLog.android	Debug Backdoor
CVE-2026-4217	ai.nreal.nebula.universal	AES Key Exposure
CVE-2026-4218	aedes.me.beta	Authorization Issue
CVE-2026-4219	ae.index.apgcs	Authorization Issue
CVE-2026-4242	app.babychakra.babychakra	Segment Misconfig
CVE-2026-4243	app.lanacion.activity	WS Credential Leak
CVE-2026-4250	albert.health	GCP Key Leak

Continued on next page

TABLE X: CVE List (Continued)

CVE ID	Package Name	Vulnerability Type
CVE-2026-4251	ai.citydata.citychat	GCP Key Leak
CVE-2026-5420	cats.goods.sort.sorting.games	AES Key Exposure
CVE-2026-5452	campusconnect.ucc	Unrestricted Upload
CVE-2026-5453	br.com.rico.mobile	Segment Misconfig
CVE-2026-5454	co.gridapp.organiser	Segment Misconfig
CVE-2026-5455	ca.diagram.dialogue	Segment Misconfig
CVE-2026-5456	com.aligntech.myinvisalign.emea	CDA Tokens Exposure
CVE-2026-5457	com.allproperty.android.agentnet	Segment Misconfig
CVE-2026-5458	com.afone.noelse	Segment Misconfig
CVE-2026-5462	com.WahooFitness.SYSTM	Segment Misconfig
CVE-2026-5471	app.investory.toyfactory	Firebase Key Exposure

B. LLM Selection

In addition to the stepwise design, the choice of LLMs in RULEDROID is crucial for ensuring the quality of the generated rules. To evaluate their impact, we compared four state-of-the-art models: GPT-5, o4-mini, DeepSeek-V3.2, and DeepSeek-R1.

Extracting Security Guidelines. For this task, we conducted experiments using a dataset of 41 files. We first manually extracted a reference set of text blocks to serve as the ground truth. Then, using identical prompts and a shared RAG knowledge base, each LLM was tasked with extracting relevant text blocks. The outputs were evaluated using the ROUGE metric. Detailed comparison results are shown in Table XI.

ROUGE-1 measures unigram overlap to evaluate basic lexical similarity. ROUGE-2 captures bigram matches to assess the consistency of key phrases. ROUGE-L, based on the Longest Common Subsequence (LCS), quantifies similarity in terms of sequence and overall coherence.

The experimental results show that for tasks involving precise text classification and information extraction, the general-purpose models GPT-5 and DeepSeek-V3.2 outperform the reasoning-focused models o4-mini and DeepSeek-R1. This advantage is due to their stronger representational capabilities, better contextual understanding, and improved generalization.

Manual inspection reveals that o4-mini often merges all security issues into a single broad category, failing to distinguish between different vulnerability types. This behavior leads to noticeable deviations from the reference text block set. DeepSeek-R1 performs slightly better than o4-mini but still tends to summarize content rather than preserving the original structure of the documentation. DeepSeek-V3.2 performs well in classification but is slightly less complete than GPT-5 in capturing knowledge related to specific security topics. Overall, GPT-5 achieves the best results by accurately extracting and segmenting text blocks with higher precision.

TABLE XI: Model performance in extracting guidelines.

Model	ROUGE-1	ROUGE-2	ROUGE-L
GPT-5	**0.7082**	**0.5613**	**0.5877**
o4-mini	0.6206	0.4342	0.4660
DeepSeek-R1	0.5932	0.3870	0.4490
DeepSeek-V3.2	0.6864	0.5084	0.5524

Evaluating Rule Applicability. In this step, we evaluate how well each model can identify security issues that are suitable for static detection. We use the text blocks extracted by GPT-5, the top-performing model from the previous task, to simulate performance under realistic conditions. A total of 72 security-related rules generated by GPT-5 were manually reviewed to create a reference set. Each of the four models was then tasked with performing the same classification task, and their outputs were compared against the reference set. The results are presented in Table XII.

TABLE XII: Model performance in evaluating rule applicability.

Model	TP	FN	FP	TN	Precision	Recall	F1-score
GPT-5	47	4	7	14	87.04%	92.16%	89.52%
DeepSeek-V3.2	45	6	6	15	88.24%	88.24%	88.24%
o4-mini	48	3	3	18	94.12%	**94.12%**	**94.12%**
DeepSeek-R1	44	7	2	19	**95.65%**	86.27%	90.72%

Initial Rule Generation and Repairing Syntax Errors. We generated detection rules using the text blocks identified by o4-mini, which achieved the best performance in the previous evaluation. The initial rules were then iteratively refined using the same model. To evaluate the rule generation capabilities of different models, we compared benchmark scanning results based on both the discarded rules (those that failed during debugging) and the successfully repaired rules. As shown in Table XIII, GPT-5 outperforms all other models in producing effective and executable detection rules.

TABLE XIII: Model performance in initial rule generation and repairing syntax errors.

Model	GPT-5	o4-mini	DeepSeek-R1	DeepSeek-V3.2
Incorrect Rules	17	25	37	18
Debug Attempts	50	87	256	56
Abandoned Rules	**0/111**	1/101	31/154	1/83
True Positive	**83**	69	62	65

Enhancing Detection Rules. We further refined the detection rules using different large language models, starting from the rule set generated by DeepSeek-V3.2, which showed the best performance in the previous phase. The refined rules were evaluated using a CVE-based benchmark to measure their effectiveness. The comparison results are shown in Table XIV.

TABLE XIV: Model performance in enhancing detection rules.

Models	GPT-5	o4-mini	DeepSeek-R1	DeepSeek-V3.2
Debug Attempts	19	25	38	36
True Positive	98	**109**	105	98

Removing Duplicate Rules. To ensure a fair comparison, we consolidated the rules generated by the best-performing model from the previous phase as a unified input. We also manually annotated duplicate rule pairs and selected the higher-quality rule from each pair to construct a reference standard. An LLM is considered correct only if it both accurately identifies duplicate rules and selects the better rule from each pair.

TABLE XV: Model performance in removing duplicate rules.

Models	GPT-5	o4-mini	DeepSeek-R1	DeepSeek-V3.2
Accuracy	93.1%	89.6%	**96.6%**	**96.6%**

As shown in Table XV, the DeepSeek-R1 and DeepSeek-V3.2 models produce the fewest deduplication errors. Given their similar accuracy, we selected DeepSeek-V3.2 for subsequent tasks due to its faster processing speed.

Classifying Rule Severity. Using a similar methodology as in the previous steps and building on the best output from the deduplication phase, we evaluated the models on their ability to classify rule severity. As shown in Table XVI, all four models achieved comparable performance, with DeepSeek-R1 attaining the highest accuracy in this task.

TABLE XVI: Model performance in classifying rule severity

Model	GPT-5	o4-mini	DeepSeek-R1	DeepSeek-V3.2
Error Number	14	**10**	**10**	12
Accuracy	86.27%	**90.20%**	**90.20%**	88.24%

False Positive Filtering. We applied the refined rule set from the previous step to the Ghera benchmark and used LLM-FPF with different models to filter false positives in the scan results. This setup allows us to evaluate each model's effectiveness in reducing false positives. The results are presented in Table XVII, where GPT-5 demonstrates a clear advantage over the other models.

TABLE XVII: Model performance in false positive filtering.

Model	TP	FN	FP	TN	FPR	Recall	F1-score
GPT-5	**33**	**26**	**1**	**58**	**1.69%**	**55.93%**	**70.97%**
DeepSeek-R1	30	29	4	56	6.78%	50.85%	64.52%
DeepSeek-V3.2	31	28	3	56	5.08%	52.54%	66.67%
o4-mini	31	28	1	58	**1.69%**	52.54%	68.13%