Towards Automated Compliance Check for IoT Apps: A Multi-Agent Approach

Yuxuan Zhang
Texas A&M University
yuz516@tamu.edu

1 Abstract

The rapid proliferation of Internet of Things (IoT) technologies has introduced significant security challenges, particularly within smart home applications. These applications, while enhancing convenience, often suffer from vulnerabilities such as cryptographic API misuse, posing serious risks to user data and system integrity. Despite the emergence of advanced static analysis tools, recent studies reveal that these methods struggle to reliably detect critical vulnerabilities, often necessitating time-consuming manual analysis by experts.

This paper presents a novel approach to automated vulnerability detection in IoT applications by leveraging large language models (LLMs). I initially developed a single-agent LLM-based system capable of detecting cryptographic misuse vulnerabilities in Java source code. However, experiments revealed a high rate of false positives, prompting the design of a multi-agent system to enhance detection reliability. The multi-agent system employs a planner agent to perform an initial analysis, assigning confidence scores to potential vulnerabilities. For high-confidence predictions, the planner directly reports vulnerabilities; for medium-confidence cases, specialized second-level agents validate the planner's findings.

The system was evaluated on nine real-world IoT applications by extracting and analyzing their Java source code. Preliminary results demonstrate the effectiveness of the multi-agent design in mitigating false positives while maintaining detection capabilities. Although challenges remain, our study highlights the potential of LLM-driven multi-agent systems as a step toward reliable and efficient vulnerability detection for IoT app compliance checks.

2 Introduction

The Internet of Things (IoT) has rapidly become a cornerstone of modern technology, with smart home devices emerging as integral components of daily life. These devices enable convenience and automation, often relying on feature-rich applications that interact extensively with cyberspace, sensors, and users. However, their complexity and reliance on sensitive data expose them to significant security vulnerabilities.[3] To mitigate these risks, IoT security regulations and standards have been established, requiring IoT applications to undergo certification by Commercially Licensed Evaluation Facilities (CLEFs). These certifications aim to ensure compliance with security regulations, including proper cryptographic practices. Despite these measures, performing compliance checks is far from trivial due to the diverse and evolving nature of vulnerabilities in IoT applications.

Recent research has highlighted critical gaps in IoT app security compliance. For instance, Mandal et al.[5] revealed that many commercially popular IoT apps fail compliance checks due to cryptographic API misuse and other vulnerabilities, such as unjustified dangerous permissions in privacy policies. Their findings also showed that state-of-the-art static analysis tools often struggle to detect these issues reliably. Worse, IoT app developers can sometimes evade detection by obfuscating vulnerable code to appear compliant. As a result, compliance checks often revert to manual analysis, which is time-intensive and requires significant expertise. This reliance on manual analysis delays the release of IoT apps, creating friction between licensing regulators and commercial entities.

To address these challenges, I propose leveraging large language models (LLMs) as a novel approach to automated vulnerability detection in

IoT applications. LLMs, such as OpenAI's GPT-4, have demonstrated remarkable potential in understanding and analyzing code due to their ability to capture semantic and contextual nuances. This capability positions LLMs as promising tools for enhancing the reliability and comprehensiveness of automated cryptographic misuse detection.

Large language models (LLMs) have shown great promise in addressing cybersecurity challenges, including code vulnerability detection, due to their ability to understand and analyze complex code structures. Prior work has explored their use in identifying vulnerabilities such as insecure cryptographic practices, injection flaws, and outdated dependencies. By leveraging their contextual and semantic understanding, LLMs can bridge the gap between traditional static analysis tools and manual reviews, enabling more automated and reliable vulnerability detection. Inspired by these advances, this work investigates the use of LLMs for detecting cryptographic API misuse in IoT app source code.

In this project, I first developed a single-agent LLM-based system, which served as a foundation for analyzing cryptographic misuse vulnerabilities. While effective in identifying potential issues, the system exhibited a high false positive rate, highlighting the need for refinement. To address this limitation, I designed a multi-agent system that combines an initial planner agent with specialized second-level agents for a more nuanced detection process. The planner identifies potential vulnerabilities and assigns confidence scores, while second-level agents provide further validation for less confident predictions. This two-tiered design reduces false positives while maintaining the system's detection capabilities. I summarize our contribution as below.

- Proposed a multi-agent system leveraging LLMs for crypto misuse vulnerability detection in Jave source code, comprising a planner and specialized second-level agents, to enhance the reliability of cryptographic misuse detection.

- Applied the proposed system to nine real-world IoT apps, with a practical workflow for converting APKs to Java code, preprocessing, and batching inputs for analysis.

- Through comparative evaluation with a single-agent system, highlighted the benefits and limitations of using multi-agent designs in automated vulnerability detection. This work provides a basis for further exploration of LLM-driven multi-agent systems for cybersecurity and IoT compliance checks.

The project is open-sourced in the following Github link: `https://github.com/michaelzhang05/Jave-Crypto-Misuse-Detection`

3 Related Work

Extend the following related work. Make sure to include your reference. Recent research highlights significant gaps in IoT security despite established compliance standards. Mandal et al.[5] explored the weaknesses in IoT app certification processes, revealing that many certified apps suffer from vulnerabilities, including cryptographic misuse and permission overreach, which current certification mechanisms often fail to detect. Arkalakis et al.[2] examined the inconsistencies in Android's Data Safety section, showing widespread misrepresentation in data collection practices, further compromising user privacy. Liu et al.[4] introduced ALGOSPEC, a novel approach that enforces purpose limitation using polynomial approximation. This method aims to ensure data is only processed with authorized algorithms, presenting a more efficient alternative to traditional cryptographic solutions. Together, these studies underline the limitations of existing automated detectors and cryptographic methods in ensuring IoT app security, highlighting the potential of large language models and new algorithmic techniques for improving vulnerability detection and compliance enforcement.
To my best knowledge, although using large language models for vulnerability detection has been widely explored by researchers, there is no complete work using large language models for modern IoT app compliance check. This project is intended to do a first attempt leveraging LLMs to resolve the lack of synthetic understanding ability problem in modern auto detectors, which is mentioned as a limitation in previous research[5].

4 Methodology

In this section, I will introduce the methodology used in this project. Section 4.1 gives the overview of the pipeline and introduce each module. Section 4.2 introduce the pre-processing module. Section 4.3 provide details about the agent design,

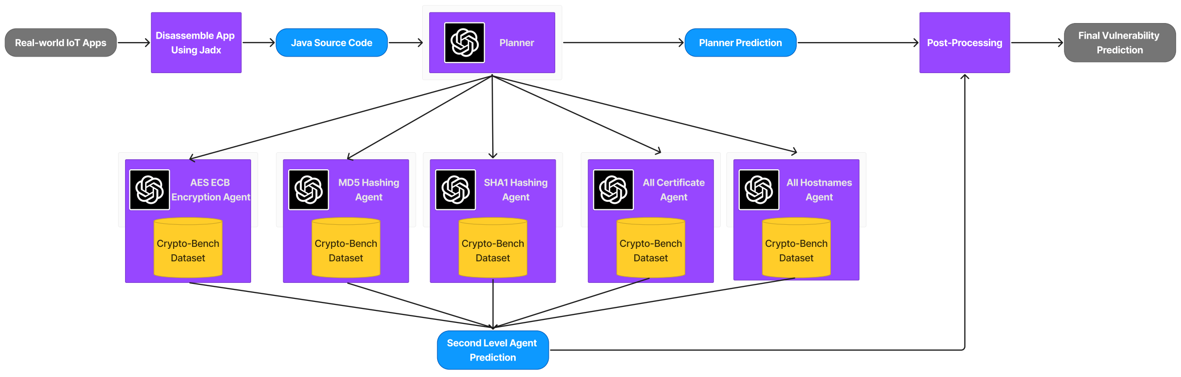

Figure 1: Method Overview

with a vanilla version of single agent, and a improved version of multi-agent system. Finally in Section 4.3, the post-processing module is presented.

4.1 Overview

Figure 1 presents an overview of the system. As Figure 1 shows, the system comprises three main stages: input preprocessing, vulnerability detection, and post-processing.

The input preprocessing module extracts Java code from Smart Home app APK files using the Jadx decompiler, filters relevant code using keyword matching, and divides it into manageable batches. (Section 4.2) The vulnerability detection module includes two designs: a single-agent system performing end-to-end detection and a multi-agent system with a planner agent and specialized second-level agents for targeted analysis based on confidence thresholds. (Section 4.3)

To provide a simple mitigation to LLM hallucination, a post-processing module aggregates and counts reported vulnerabilities across batches. By applying a mean-based sampling approach, it filters out less reliable predictions to make a more robust final report. (Section 4.2)

4.2 Pre-Processing

The input preprocessing stage prepares Java source code for analysis by the vulnerability detection modules. This process begins with extracting Java code from Smart Home application APK files using the Jadx decompiler[6]. The raw decompiled code often contains a significant amount of irrelevant or auxiliary code unrelated to cryptographic operations. To focus the analysis, I apply a keyword-matching algorithm to retain only the code segments likely to involve cryptographic API usage, such as those containing references to classes like Cipher or MessageDigest.

After filtering, the relevant code is divided into manageable chunks to accommodate the input size limitations of LLMs. Each chunk is formatted into a batch, ensuring that no single batch exceeds the token limit of the LLMs. This batching process is designed to maximize the efficiency of the vulnerability detection modules while maintaining the integrity of the extracted code context.

4.3 Agent Analysis

1. Single Agent

The single-agent system leveraged OpenAI's GPT-4o-mini model to identify vulnerabilities based on a structured prompt. The prompt described 10 common cryptographic misuse vulnerabilities, such as ”AES with ECB for encryption,” ”No cleanpassword() call after PBEKeySpec”, ”MD5 hashing,” and ”Trusting all certificates”, as illustrated in Listing 1. Each vulnerability included a brief description and a code example, ensuring the LLM had sufficient context to recognize these patterns in the input code. Also, the agent was instructed to respond in a highly specific format, either listing the vulnerabilities detected along with relevant code snippets or explicitly stating ”None Vulnerability Found” if no issues were identified. This structured interaction ensured consistency in the model's outputs.

```
"""
2. AES with ECB for encryption
Description: The code uses AES with ECB
    (Electronic Codebook) mode, which is
     insecure due to deterministic
    output for identical plaintext.
Example:
```

```
Cipher cipher = Cipher.getInstance("AES/
    ECB/PKCS5Padding");

4. No cleanpassword() call after using
    PBEKeySpec
Description: The code fails to call `
    cleanPassword()` after using `
    PBEKeySpec`, which can leave
    sensitive data in memory.
Example:
PBEKeySpec keySpec = new PBEKeySpec(
    password);
// Missing keySpec.clearPassword();

5. MD5 hashing
Description: The code uses the MD5
    hashing algorithm, which is
    considered insecure and prone to
    collisions.
Example:
MessageDigest md = MessageDigest.
    getInstance("MD5");

7. Trusting all certificates
Description: The code trusts all SSL
    certificates, which exposes the
    system to man-in-the-middle attacks.
Example:
TrustManager[] trustAllCerts = new
    TrustManager[] {{
     new X509TrustManager() {{
         public void checkClientTrusted(
    X509Certificate[] certs, String
    authType) {{}}
         public void checkServerTrusted(
    X509Certificate[] certs, String
    authType) {{}}
         public X509Certificate[]
    getAcceptedIssuers() {{ return null;
      }}
     }}
}};
"""
```

Listing 1: Single-Agent Vulnrability Definition

Limitation of the Single Agent Approach: The single-agent design serves as an initial attempt to leverage LLM for vulnerability detection. It is simple and straightforward, resulting in obvious limitations. A major challenge was the high false positive rate (0.87 without post processing as shown in Table 1). The system often misinterpreted benign code structures as vulnerabilities due to the broad generalization of the provided examples. For instance, code that resembled cryptographic misuse patterns but was semantically secure was frequently flagged as vulnerable. Additionally, the model struggled to handle vulnerabilities requiring nuanced domain-specific expertise, such as distinguishing between secure and insecure implementations of AES encryption.

The single-agent system's performance also varied across different batches of input, with inconsistent predictions observed for similar code segments. Furthermore, the reliance on a single decision-making process limited the system's scalability and reliability, which is critical considering the real-world apps considered in this project contains large amount of code (more than 17,000 files).

4.4 Multi-Agent

The multi-agent system refines the single-agent design by addressing its high false positive rate and lack of nuanced validation. This hierarchical approach integrates a planner agent for initial analysis and specialized second-level agents for deeper investigation of specific vulnerabilities, providing a balance between general-purpose detection with domain-specific expertise.

As illustrated in Figure 1, the planner performs initial vulnerability detection. Other than the predicted vulnerabilities, the planner is required to assign a confidence score for each of the vulnerability it predicts, as illustrated in Listing 2. The high-confidence predictions in the final report. Medium-confidence predictions trigger second-level agents, which validate the planner's findings with more precise analysis. These second-level agents, trained using examples from the Crypto-Bench dataset[1], specialize in specific vulnerabilities like MD5 hashing or AES with ECB encryption, providing binary outputs: either confirming the vulnerability or marking the code as secure. For each of the second level agent, few-shot prompting is used to provide domain specific knowledge that guide the agent to perform analysis on specific vulnerabilities.

```
"""
If the code is vulnerable, respond with:
"Vulnerable"
Followed by a list of detected
    vulnerabilities and their likelihood
     scores (e.g., 1: 0.85, 3: 0.90).
Only include the detected
    vulnerabilities and scores in your
    response. Do not list the undetected
     vulnerabilities.
Example:
Vulnerable
1: 0.75
5: 0.90

If the code is not vulnerable, respond
    with:
"None Vulnerability Found"
"""
```

Listing 2: Planner Response Definition

Table 1: Overall Performance

Model	**Accuracy**	**Precision**	**Recall**	**F1**	**FPR**	**FNR**
Single-Agent (w/o Post-Processing)	0.47	0.78	0.47	0.37	0.87	0.00
Single-Agent (w/ Post-Processing)	0.73	0.73	0.73	0.72	0.13	0.49
Multi-Agent (w/o Post-Processing)	0.53	0.58	0.53	0.54	0.55	0.34
Multi-Agent (w/ Post-Processing)	0.77	0.77	0.77	0.77	0.20	0.29

Table 2: Misuse Cases Across IoT Applications (Human Expert Analysis)[5]

No.	**Misuse Case**	**Tuya**	**NetHome Plus**	**Eureka**	**Midea Air**	**MSmartHome**	**GreenMAX**	**Wyze**	**Dals Connect**	**Google Home**	**Total**
1	Only AES for encryption	✓									1
2	AES with ECB for encryption	✓				✓		✓	✓		4
3	AES with CBC for encryption	✓				✓	✓	✓	✓	✓	6
4	No `clearpassword()` call after using PBEKeySpec						✓				1
5	MD5 hashing	✓	✓	✓	✓	✓			✓	✓	7
6	SHA1 hashing		✓	✓	✓	✓			✓	✓	6
7	Trusting all certificates					✓		✓		✓	3
8	Allowing all hostnames					✓		✓		✓	3
9	“SSL” as context							✓	✓		2
10	“TLSv1” as context	✓								✓	2
Total		5	2	2	2	6	2	5	5	6	35

The final report integrates results from the planner and second-level agents. High-confidence predictions are accepted directly, while medium-confidence ones are validated by second-level agents. For vulnerabilities lacking a second-level agent, the system relies solely on planner confidence.

4.5 Post-Processing

The post-processing module is added to provide a simple mitigation of the hallucination problem of the LLM. Both the single-agent and multi-agent systems sometimes report false positives or inflate their predictions due to overconfidence or misinterpretation of the code. To address this, the post-processing module aggregates the vulnerabilities detected across all analyzed batches and performs a statistical filtering process. Specifically, it counts the number of times each vulnerability is reported and applies a threshold-based mechanism—such as mean or median filtering—to identify the most consistently reported vulnerabilities as the final predictions. This approach ensures that rare or spurious reports are excluded, improving the reliability and precision of the results.

5 Evaluation

5.1 Implementation

In this project, the agents are implemented using the LangChain framework. OpenAI GPT-4o-mini is selected as the foundation LLM for all the agent, including the second level agents in the multi-agent system. The single-agent system is implemented in 256 lines of Python code while the multi-agent system is implemented in 1,014 lines of Python code. Among the 10 vulnerabilities, 5 of them have correspondingly designed second level agent to perform further investigations.

5.2 Dataset

To build the second level agents, examples of vulnerabilities are provided as domain specific knowledge. The knowledge are selected from an open-source dataset from previous work, called Crypto-API Bench[1]. The Crypto-API Bench dataset is a comprehensive benchmark designed to evaluate cryptographic API misuse in Java code. It includes 16 distinct types of cryptographic vulnerabilities, with a mixture of both secure and insecure code snippets for each vulnerability.

5.3 Experiment Setup

Experiments were run on my personal laptop, using a Apple M1 chip, with 32 GB RAM. The real-world Smart Home apps are chosen as the ones investigated by human experts in previous work[5], for a easy comparison of our approach to the human expert analysis results, and also the state-of-art automated detection tools based on static analysis. The human-expert analysis results from previous work is considered the ground truth, as listed in Table 2.

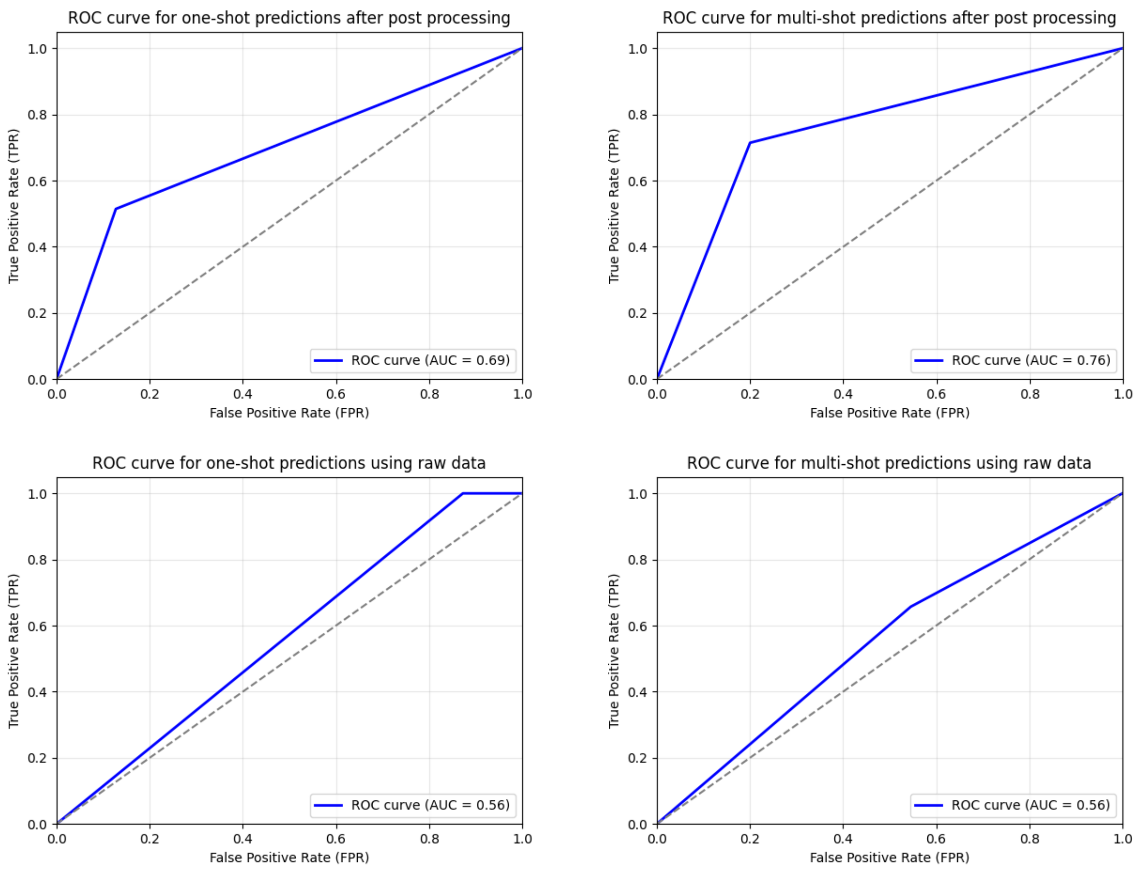

Figure 2: ROC Curve

5.4 Results

1. Overall Performance

Table 1 demonstrates the effectiveness of the multi-agent system in reducing false positives while maintaining robust detection performance. The single-agent system, without post-processing, suffers from a high false positive rate (FPR) of 0.87 despite achieving a precision of 0.78. Adding post-processing improves its accuracy and recall to 0.73 but introduces a higher false negative rate (FNR) of 0.49.

In contrast, the multi-agent system significantly lowers the FPR to 0.55 without post-processing and further reduces it to 0.20 when post-processing is applied. With post-processing, the multi-agent system achieves the highest overall performance, with accuracy, precision, and recall all at 0.77. These results highlight the multi-agent system's ability to improve reliability and reduce false positives compared to the single-agent design.

Figure 2 presents the ROC curve for the 4 models listed in Table 1. As the Figure shows, the multi-agent system has a higher AUC, indicating better performance compared to the single-agent design.

2. Comparison to Human Expert Analysis Compared to human expert analysis, which identified 35 vulnerabilities across the 9 tested apps, the LLM-based approaches demonstrated varying levels of performance. The single-agent system detected 17 out of the 35 vulnerabilities, while the multi-agent system improved significantly, detecting 25 vulnerabilities. These results indicate the value of incorporating a hierarchical design with specialized second-level agents in the multi-agent system. However, the performance of the LLM-based systems still falls short compared to existing automated tools like CogniCrypt and MobSF, which identified 33 and 28 vulnerabilities, respectively. While LLMs bring flexibility and adaptability, these results highlight the need for further refinement to match the consistency and coverage of traditional static analysis tools.

Table 3: Misuse Cases Across IoT Applications (Single Agent)

No.	Misuse Case	Tuya	NetHome Plus	Eureka	Midea Air	MSmartHome	GreenMAX	Wyze	Dals Connect	Google Home	Total
1	Only AES for encryption										0
2	AES with ECB for encryption										0
3	AES with CBC for encryption	✓								✓	2
4	No `clearpassword()` call after using PBEKeySpec										0
5	MD5 hashing	✓	✓	✓	✓	✓	✓	✓	✓	✓	9
6	SHA1 hashing	✓	✓	✓	✓	✓	✓	✓	✓	✓	9
7	Trusting all certificates					✓					1
8	Allowing all hostnames					✓					1
9	"SSL" as context					✓				✓	2
10	"TLSv1" as context										0
Total		2	2	2	2	5	2	2	2	4	18

Table 4: Misuse Cases Across IoT Applications (Multi Agent)

No.	Misuse Case	Tuya	NetHome Plus	Eureka	Midea Air	MSmartHome	GreenMAX	Wyze	Dals Connect	Google Home	Total
1	Only AES for encryption	✓							✓	✓	1
2	AES with ECB for encryption					✓	✓	✓			4
3	AES with CBC for encryption	✓	✓						✓	✓	6
4	No `clearpassword()` call after using PBEKeySpec						✓				1
5	MD5 hashing	✓	✓		✓	✓		✓			7
6	SHA1 hashing	✓	✓		✓	✓			✓	✓	6
7	Trusting all certificates			✓		✓		✓		✓	3
8	Allowing all hostnames					✓	✓	✓		✓	3
9	"SSL" as context				✓				✓	✓	2
10	"TLSv1" as context									✓	2
Total		4	3	1	3	5	3	4	5	7	35

6 Conclusion

6.1 Discussion

This project explores the use of large language models for automated detection of cryptographic API misuse in Java source code, trying to provide reliable automated censorship investigation for Smart Home apps. In this project, I developed a single-agent system and a multi-agent hierarchical design. The multi-agent system demonstrates clear advantages, improving both detection accuracy and reliability by combining general-purpose analysis from the planner with domain-specific expertise from second-level agents. The inclusion of post-processing provides attempt to mitigate the hallucination problem, ensuring that final predictions are more robust. Results show that the multi-agent system detects 25 out of 35 vulnerabilities identified by human experts, a significant improvement over the single-agent system's 17 vulnerabilities. However, while the multi-agent design reduces false positives, it still lags behind existing automated tools like CogniCrypt (33/35) and MobSF (28/35), highlighting a trade-off between flexibility and domain-specific optimization. Nonetheless, the multi-agent system's modular and scalable design offers a promising framework for enhancing LLM-based vulnerability detection.

6.2 Limitation and Future Work

Despite its advancements, the proposed system has limitations. First, it relies heavily on confidence thresholds and handcrafted prompts, which may not generalize well across different datasets or programming domains. The second-level agents, although effective, are limited to only five specific vulnerabilities, leaving gaps for vulnerabilities without corresponding agents. Additionally, the LLM-based systems underperform compared to state-of-the-art static analysis tools in terms of overall detection coverage. The system also faces challenges with computational overhead, as the multi-agent design requires multiple rounds of analysis, increasing runtime. Finally, while post-processing reduces hallucinations, its mean-based filtering may discard rare but valid vulnerabilities, suggesting a need for more nuanced filtering techniques. These limitations underscore the need for further refinement and optimization to make LLM-based vulnerability detection systems both practical and competitive.

References

[1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. “CryptoAPI-Bench: A comprehensive benchmark on Java cryptographic API misuses”. In: *2019 IEEE Cybersecurity Development (SecDev)*. IEEE. 2019, pp. 49–61.

[2] Ioannis Arkalakis et al. “Abandon All Hope Ye Who Enter Here: A Dynamic, Longitudinal Investigation of Android’s Data Safety Section”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 5645–5662.

[3] Xin Jin et al. “Understanding iot security from a market-scale perspective”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1615–1629.

[4] Shuofeng Liu et al. “Being Transparent is Merely the Beginning: Enforcing Purpose Limitation with Polynomial Approximation”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 6507–6524.

[5] Prianka Mandal et al. “” Belt and suspenders” or” just red tape”?: Investigating Early Artifacts and User Perceptions of {IoT} App Security Certification”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 4927–4944.

[6] skylot. *JADX: Dex to Java Decompiler*. Accessed: 2024-10-05. 2024. URL: `https://github.com/skylot/jadx`.