Automatically Generating Rules of Malicious Software Packages via Large Language Model

XiangRui Zhang*, XueJie Du*, HaoYu Chen*, Yongzhong He*, Wenjia Niu*, Qiang Li*†

*School of Cyberspace Security, Beijing Jiaotong University, China

Abstract—Today's security tools predominantly rely on predefined rules crafted by experts, making them poorly adapted to the emergence of software supply chain attacks. To tackle this limitation, we propose a novel tool, RULELLM, which leverages large language models (LLMs) to automate rule generation for OSS ecosystems. RULELLM extracts metadata and code snippets from malware as its input, producing YARA and Semgrep rules that can be directly deployed in software development. Specifically, the rule generation task involves three subtasks: crafting rules, refining rules, and aligning rules. To validate RULELLM's effectiveness, we implemented a prototype system and conducted experiments on the dataset of 1,633 malicious packages. The results are promising—RULELLM generated 763 rules (452 YARA and 311 Semgrep) with a precision of 85.2% and a recall of 91.8%, outperforming state-of-the-art (SOTA) tools and scored-based approaches. We further analyzed generated rules and proposed a rule taxonomy: 11 categories and 38 subcategories.

I. INTRODUCTION

Open source software (OSS) has become a cornerstone of software development, enabling developers to build applications efficiently by reusing codebases, libraries, and tools. Meanwhile, OSS ecosystems are suffering new security challenges, including malware infiltration, supply chain attacks, and vulnerabilities in third-party dependencies. Attackers can exploit vulnerabilities in popular packages, distribute malware, or insert malicious code into legitimate software. Recently, the number of security incidents in OSS ecosystems has increased significantly, with an annual growth rate of 742% in the past five years [1, 2, 3], e.g., 0 incidents in 2018, 249 incidents in 2019, and 690,211 incidents in 2023. For instance, LOG4J vulnerabilities [4] brought great risks to software systems and third-party packages relying on this library. This attack propagated across downstream projects, potentially compromising thousands of applications, and underscoring the systemic risks inherent in the OSS ecosystem.

To address these risks, developers and security analysts rely on software detection tools to discover underlying risks in OSS ecosystems. Most software detection tools, such as SemGrep [5], YARA [6], and AppInspector [7], predominantly rely on malware signatures and predefined rules. Those tools are designed for pattern-matching malware and analyzing suspicious files based on textual or binary features. In particular, predefined rules leverage built-in specific patterns, strings, and features to identify underlying threats. However, crafting detection rules requires manual effort and domain

knowledge, which is not scalable for large and diverse software packages in OSS ecosystems. Moreover, as the number of malware grows, manual written rules struggle to keep pace with emerging threats, lacking adaptability in software supply chain (SSC) attacks.

In this work, we introduce large language models (LLMs) to automate rule generation, serving as a supplement to existing security tools. Nowadays, the LLM offers an innovative approach to enhance security detection and analysis capabilities, including vulnerability identification [8] and repair [9, 10], static analysis [11, 12], and reverse engineering [13]. For example, Li et al. [11] investigated LLMs' capabilities in static analysis for automatically repairing zero-shot vulnerabilities, and Wang et al. [10] explored project-level vulnerability detection via LLMs (e.g., ChatGPT or CodeLlama). In contrast, we propose leveraging LLMs to enhance security tools' capabilities by automating the creation of rules that are tailored to detect both known and emerging threats.

However, directly utilizing LLMs for rule generation poses three technical challenges. First, many malicious packages share similar code bases, leading to redundancy and inaccuracies in the generated rules. Second, LLMs struggle to process the extensive source code of many malicious packages, which may exceed their input limitations. Third, due to the inherent generation characteristics of LLMs, such as hallucinations, they may produce rules that are unsuitable for direct use in development environments. Additionally, some malware employs obfuscation techniques to conceal its intent, further complicating detection.

To address those challenges, we propose a novel tool, RULELLM, designed to automatically generate YARA & Semgrep rules for detecting risks in OSS ecosystems. Rather than relying solely on LLM, RULELLM decomposes rule generation into different subtasks: crafting, refining, and aligning rules. In the crafting stage, RULELLM extracts independent code blocks from malware and creates coarse-grained rules based on these blocks. In the refining stage, redundant or ineffective rules are merged, ensuring higher specificity and accuracy. To mitigate the risk of hallucinations of LLMs, RULELLM integrates a specialized LLM-based agent to refine rules. The agent's feedback loop allows for dynamic refinement of the generated rules. The outputs of RULELLM include YARA and Semgrep rules that are ready for direct deployment in security workflows.

To validate the effectiveness of RULELLM, we implemented a prototype system based on several open-source

†Qiang Li is the corresponding author; email: liqiang@bjtu.edu.cn

libraries. Our dataset comprised 3,200 malware packages collected from GuardDog [14], which were reduced to 1,633 unique packages after deduplication. We also selected 500 of the most commonly used legitimate packages from [15]. We benchmarked RULELLM against multiple baselines, including state-of-the-art (SOTA) tools, scored-based approaches, and several diverse LLMs (GPT-3.5, GPT-4o, Llama, Claude). Experimental results demonstrate that RULELLM generated 763 rules (452 YARA and 311 Semgrep) with a precision of 85.2% and a recall of 91.8% in identifying malicious packages, significantly outperforming the baseline tools. Notably, the generated rules are fully compatible with existing systems and can be directly deployed to scan software packages without errors. Furthermore, we conducted a systematic analysis of the generated rules and proposed a comprehensive rule taxonomy. The taxonomy consists of 11 categories and 38 subcategories, offering insights into the characteristics and applications of the rules.

In short, our contributions are as follows:

- We have proposed RULELLM, a novel tool [16] that automatically generates YARA and Semgrep rules for OSS ecosystems.
- We have generated 452 YARA and 311 Semgrep rules that are well-formatted and can be directly deployed in existing tools. Experimental results demonstrated that RULELLM outperforms SOTA tools.
- We have released RULELLM's tool and 763 compatible rules to the research community, as a supplement to security detection tools.

Roadmap. The rest of the paper is organized as follows. Section II illustrates the background of LLM and rule-based detection tools. Section III and IV present the design of RULELLM, including the malware information extraction and rule generation. Section V evaluates RULELLM on real-world malicious packages and compares it with existing tools. Section VI discuss the limitations of RULELLM and its generated rules. Section VII discusses related work, and Section VIII concludes the paper.

II. BACKGROUND

In this section, we provide background about LLMs, YARA/Semgrep rules, and the technical challenges.

A. Large Language Model

Large Language Models (LLMs) [17, 18, 19, 20] acts as a generation model that outputs a sequence of tokens to complete given input sequences. For instance, an LLM can generate a sequence of prediction tokens (an answer) in response to a sequence of input tokens (a question). Tokens refer to common characters with a unique numeric identifier, up to a pre-training corpus. The input to an LLM is typically structured as a prompt, which defines the sequence of input tokens. Different prompts (inputs) can lead to diverse capacities of LLM and determine its effectiveness across various tasks. Hence, researchers and developers use prompt engineering to

```
rule base64 : base64{
    meta:
        description = "Base64 encoded blob"
    strings:
        $a = /([A-Za-z0-9+/]{4}|\{2,}([A-Za-z0-9+/]{3})=)?\b/
    condition:
        $a
    }
```

```
rules:
 - id: detect-torrent-client-info-retrieval
   languages: [python]
   message: "Detected torrent client.........."
   patterns:
    - pattern: |
        $CLIENT.torrents_info(torrent_hashes=$HASH)
   metadata:
     Detect torrent client info retrieval
```

TABLE I: The upper part is a YARA rule, and the below part is a Semgrep rule.

design robust and effective prompts to improve the capacity of LLMs on a wide range of complex tasks.

Here, we illustrate several techniques in prompt engineering for LLMs. (1) *Task decomposition* is to break down a larger task into the large, complex task into smaller, manageable subtasks. Then, LLMs execute each subtask sequentially to ensure the completion of the entire task. (2) *Chain-of-thought* introduces intermediate reasoning steps to execute a task, enabling LLMs to "think step by step" [21]. (3) *Tree-of-thought* generates multiple thoughts at each step, forming a decision tree. Then, LLMs use search algorithms, such as breadth-first search (BFS) or depth-first search (DFS), to navigate and execute the task [22]. (4) *Self-reflection* is to use LLMs to evaluate their own prior outputs to provide feedback and improve subsequent results. For instance, previous works [23, 24] leverage human feedback, such as error corrections, to fine-tune LLMs for enhanced performance. In short, prompt engineering is about obtaining the "best" results of LLMs when we don't retrain the model.

In this paper, we propose leveraging prompt engineering techniques to enable LLMs to automatically generate rules for security detection tools.

B. YARA & Semgrep Rule

Today's security tools use rule-based detection manner to discover potential vulnerabilities, malware, and security threats. Tools leverage these rules to scan OSS packages and to find suspicious entities, such as specific file structures, suspicious functions, unusual network activity, or indicators of insecure configurations.

YARA & Semgrep rules are two widely used rule formats in cybersecurity, both employing pattern-matching techniques to detect malicious behaviors and threats. A YARA rule is a structured, text-based format with a '*.yar' file extension. It typically consists of four main components: a rule name, a 'meta' section, a 'Strings' section, and a 'Condition' section.
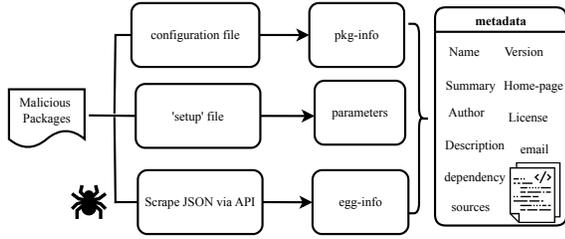
Fig. 1: Extracting the metadata of the software package.

Fig. 2: Extracting the package code.

In contrast, a Semgrep rule is written in YAML format with a '*.yaml' file extension and comprises four key components: a 'metadata' section, a 'pattern' section, a 'languages' section, and a 'message' section. Table I illustrates two examples of YARA and Semgrep rules. The YARA rule detects base64-encoded blobs, while the Semgrep rule identifies torrent information retrieval. Both rule formats are designed for clarity and efficiency, offering robust mechanisms for integrating security checks into development workflows.

However, rule-based detection approaches face several limitations. First, writing a YARA&Semgrep rule relies on expert knowledge and manual effort, making it challenging to keep up with the rapidly evolving threat landscape. Second, YARA&Semgrep rules are not specifically designed for OSS ecosystems. YARA rules focus on identifying patterns in files and binaries, whereas Semgrep rules analyze structured source code. Our investigation shows 4,574 YARA and 2,841 Semgrep rules, with most related to email, cloud, mobile, and APT attacks. Only 380 rules (46 YARA and 334 Semgrep) are related to OSS packages. Third, the growing prevalence of malware and supply chain attacks in OSS ecosystems has made it increasingly challenging to develop rules that comprehensively address all possible risks.

C. Technical Challenges

Directly leveraging LLM to generate rules presents three key technical challenges, outlined as follows:

- Malicious packages frequently exhibit diverse behaviors, including privilege escalation, information leakage, code obfuscation, and remote network operations, further complicating accurate rule creation. Additionally, attackers may use anti-analysis techniques to conceal malicious behavior, making it difficult for LLMs to capture features.
- The LLM has a context length limitation that determines the maximum volume of information of LLMs' prompts. Many malicious packages have several source code files with token counts that exceed the LLM's context length, resulting in incomplete analyses.
- Due to the strict syntax and structural requirements of the rules, LLM may involve errors and hallucinations when generating YARA and Semgrep rules.

To address these technical challenges, we propose RULELLM, a tool that automatically generates YARA and Semgrep rules without any manual effort. The generated rules can be directly utilized in existing security tools.
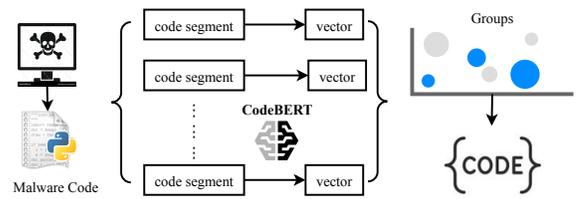
III. MALWARE KNOWLEDGE EXTRACTION

In this section, we present the methodology that automatically extracts metadata and code snippets from a malicious package.

A. Package Metadata

Package metadata is information that package authors maintain within the project. Specifically, metadata lists dependencies, URLs, versions, package names, and descriptions, which are useful for finding and installing packages from OSS ecosystems. There are 3 manners to extract its metadata. Figure 1 depicts the process of the metadata extraction, including the 'egg-info', 'package-info', and 'setup' files. (1) The 'pkg-info' file is similar to the project's configuration file, which describes the package's installation and use. (2) The 'setup' file defines the metadata and configuration for a package. This file is essential for making the package installable via the OSS ecosystem. (3) The 'egg-info' contains the descriptive information of the package on the OSS ecosystem, which provides an API endpoint for retrieving package metadata. For instance, an API endpoint from the NPM ecosystem is shown as https://registry.npmjs.org/{package_name}, and its response is a JSON file for the package metadata. Given a malicious package, we extract its metadata in those ways. Note that we use the package metadata as the LLM's input rather than the original package.

The rationale behind this is that the metadata of the malicious package is different from the benign package. First, malicious packages may have minimal, fake, or no author information, but benign packages usually have valid author names, emails, and sometimes links to their profiles or organizations. Second, malicious packages often have excessive or unusual dependencies, sometimes including outdated or obscure packages that aren't typically used in benign software. Third, malicious packages often use typo-squatting or lookalike names (e.g., 'reqests' instead of 'requests'). Descriptions may be vague or directly copied from the legitimate package to mislead users. In contrast, benign packages usually have clear, unique names and detailed descriptions. Hence, rules may be associated with the package metadata, and RULELLM generates the detection rules based on malicious packages' metadata.

B. Malicious Code Snippets

Malicious packages often contain code specifically crafted to perform unwanted or harmful actions on a system. For
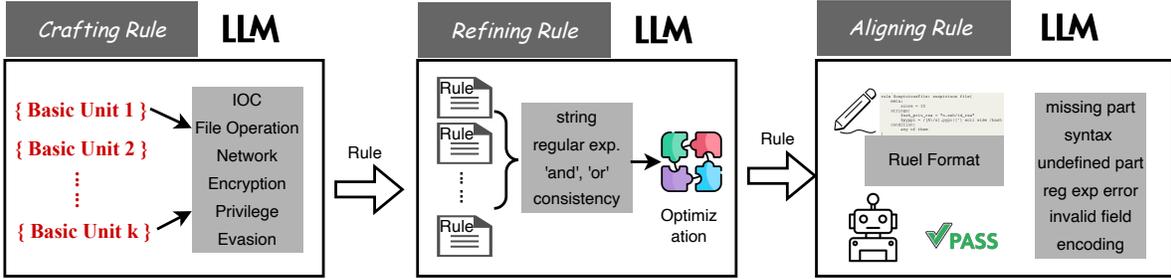
Fig. 3: RULELLM: the architecture of the LLM-based rule generation: (1) crafting basic-unit rules, (2) refining rules, and (3) aligning rules.

example, an unusual 'setup.py' or post-install command that executes unwanted code during package installation. We propose to extract distinguished code snippets from the malicious package to represent the malicious behavior, as shown in Figure 2.

Unpacking. To obtain malicious code, we first need to download the package and unpack it to a folder. Unpacking refers to extracting a package's contents from a compressed or archived format into usable files and folders. This operation involves various tasks, such as decompression, file extraction, and directory creation. After unpacking a software package, we obtain the source code files, e.g., the file in the PyPI ecosystem uses the extension '.py', while that in the NPM ecosystem uses the extension '.js'.

Vectorization. For the source code, we leverage embedding technical to convert source code into numerical vectors. The embedding represents the content and characteristics of source code in a real-valued continuous vector space. We use lexical analysis techniques to process the source code: splitting code segments, numerical representation, and concatenation. (1) Splitting. The source code S_{code} is divided into code segments. For example, $S_{code} = \{code^1, code^2, ..., code^n\}$, where $code^i$ presents a code segment, and n is the number of code segments. Note each code segment has a fixed length, denoted as a threshold. (2) Numerical representation. The pre-trained model converts each code segment into numerical representations, which contain information about the structure and semantics of the code. The formula is $v^i = f(code^i)$, where v^i presents a vector of a code segment. (3) Concatenation. We concatenate all numerical representations into a single vector, where $V_{code} = \{v^1, v^2, ..., v^n\}$ represents the source code S_{code}. Specifically, we use 512 as the length threshold to split the source code. The pre-trained model is CodeBert [25], and we use the NumPy [26] to join all numerical arrays into one vector.

Group. We utilize a clustering algorithm to group similar malware code snippets into the same cluster. The similarity between two code snippets is measured using the Euclidean distance in vector space. For implementation, we employ the Scikit-learn library [27] to apply the K-Means algorithm. In the initialization phase, we set the random seed for the centroids to 42 and define the maximum number of iterations

for the K-Means algorithm as 500. Clusters with an intra-similarity below 0.85 are discarded, as they lack sufficient homogeneity. Conversely, clusters with an intra-similarity of 0.85 or higher are retained. After clustering, the similarity among code snippets within a cluster is high, whereas the similarity between different clusters is minimal. RULELLM generates rules by analyzing and synthesizing code snippets within each retained cluster.

IV. RULE GENERATION

Our target is to generate YARA & Semgrep rules that can be seamlessly integrated into existing security tools and effectively detect threats without errors. Figure 3 presents the architecture of RULELLM, which utilizes an LLM to automate rule generation. The rule generation is divided into three subtasks: crafting rules, refining rules, and aligning rules. (1) This component involves randomly selecting several basic units from code or metadata groups. In terms of those basic units, the LLM crafts rules to cover possible features and patterns associated with malicious packages. (2) The LLM audits all coarse-grained rules from the first subtask. Then, the LLM merges rules into a scalable and effective rule. (3) An LLM-based agent determines whether the generated rule can pass the compilation process. The agent is equipped with two creation functions that compile rules. If a rule successfully compiles, we output the finalized rule. If a rule fails, the compiler outputs error messages, and the agent leverages error messages as the guideline to fix rules. In the following, we elaborate on the details of those components in RULELLM.

A. Crafting Rules

Basic Unit We divide the metadata and code of malware into basic units, which serve as the foundation for creating coarse-grained rules.

(1) Metadata is extracted as a JSON format as described in Section III-A. The entire metadata of a package is treated as a base unit. Several metadata attributes may indicate malicious behaviors, including malicious dependency libraries, empty descriptions, zero versions, and typosquatting. Table II lists four audit categories for package metadata. ① Empty information: the package has an empty description. ② Release zero: the package has a version like 0.0 or 0.0.0. ③ Typosquatting:

4

TABLE II: Identifying malicious behaviors from basic units.

Metadata	Description
Empty information	an empty description
Release zero	a version like 0.0 or 0.0.0
Typosquatting	similar name to a popular package
Dependencies	malicious dependency libraries

Code	Description
IOC	Identify compromised indicators or technical behaviors in code segments
File	Find file operations like 'open()', 'write()', 'remove()' or suspicious file paths
Network	Detect API calls or requests for C2 server connections or data exfiltration
Encryption	Identify the use of encryption algorithms like 'AES', 'RSA', or base64 encoding
Privilege	Identify privilege escalation like 'setuid()', 'setgid()' or 'CreateProcess()'
Anti-debug /Anti-analysis	Detect functions for debuggers, sandbox environments or VM detection techniques

TABLE III: The prompt in the LLM: instructions are used in the system role; Blue indicates user input; Orange indicates the (YARA|SemGrep) rule example; {...} indicates omitted content due to the page limitation.

Prompt on rule generation from the basic unit.

System role is as follows:
Task. As a senior malware code analyst, please analyze the following code samples from the same malware cluster and design effective {YARA|SemGrep} rules. These samples are variants from the same malware family.
Sample 1: {user input}
Sample 2: {user input}
Thought Process:
1. Initial Analysis: { ... ... }
2. In-depth Analysis: { ... ... } *refer to Table II*
3. External Knowledge Analysis: { ... ... }
4. Understanding and Validation: { ... ... }
Output.
1. Analysis Result {*.txt format}
2. Write {YARA|SemGrep} rules based on the analysis result.

User's information is as follows:
Input: {Basic Unit One}
Input: {Basic Unit Two}
Few Shot: {rule file}

---

the package has a similar name to a popular package. ④ Dependencies: the package has malicious dependency libraries. We only focus on the suspicious parts of the metadata.

(2) Code is organized into different groups as described in Section III-B, where snippets within the same group exhibit high similarity. However, due to the LLM's input length limitation and the complexity of the code, entire code snippets cannot be directly processed. Each code snippet is divided into multiple basic units to address this. A basic unit represents a code block: a module, a function body, and a class definition. Based on the definition provided in the Python documentation [1], our extraction process follows these steps: ① Use regex to identify whether the code begins with a specific string (e.g., 'def ', 'class ', 'if ', 'for ', 'while ', 'try:', 'with ', :); ② Add the following code to the basic unit; ③ Continue adding code until the next matched string is found; ④ Extract a new basic unit if its size exceeds 4000 characters. Each basic unit is self-contained and encapsulates specific behaviors or functionalities of the package. This division ensures that: the length of each basic unit is manageable for the LLM; the code complexity is reduced; and each unit remains meaningful for rule generation. Breaking down the code into basic units enables the LLM to efficiently analyze and generate rules without being constrained by input length or complexity.

The LLM audits the code snippet to determine whether the package exhibits potential malicious places, as outlined in Table II. ① Indicators of Compromise (IoC): Information that indicates a high probability of unauthorized access to the system, such as DNS requests or IP addresses. ② File Operation: Detection of suspicious file read and write operations within the code. ③ Network Activity: Identification of API calls or requests made to malicious servers or IP addresses. ④ Encryption Function: Detection of functions used for evasion or obfuscation techniques within the code. ⑤ Privilege Operation: Identification of operations related to privilege escalation.

⑥ Anti-debug/Anti-analysis Operation: Detection of functions designed to prevent sandbox environments or debuggers.

Multiple Similar Units. After partitioning, the LLM audits multiple similar basic units irather than a single one. This approach ensures that the generated rules are scalable and general, avoiding reliance on specific implementation details such as hardcoded strings or individual files. Several similar units are chosen from the same group. While these units may exhibit slight differences in their code blocks, the LLM identifies and extracts common behaviors and features. This strategy enhances the rule's ability to generalize across various malicious patterns, improving its effectiveness and adaptability.

Prompt plays an important role in guiding the LLM to generate the expected results through a series of instructions. Table III lists the prompt used for the basic unit rule creation. The prompt follows the Chain-of-Thought (CoT) methodology, which divides the task into a series of linear steps. Specifically, the task is divided into 4 steps: initial analysis, in-depth analysis, external knowledge analysis, and validation. (1) Initial analysis: Perform a code audit on the basic unit and provide a summary of the code. (2) In-depth analysis: Extract features or strings from the code based on the criteria listed in Table II. (3) External knowledge analysis: Determine whether the input matches known malicious behavior patterns, such as worm propagation, ransomware encryption, or remote command execution. If a match is identified, existing patterns are leveraged to construct a rule. (4) Validation: Ensure reasoning consistency and confirm that the rule covers the potential behaviors exhibited by the code.

Output. This component in RULELLM produces two outputs. The first output is a detailed analysis result, saved in the '*.txt' format. This file provides a summary of the insights into

---

[1] https://docs.python.org/3/reference/executionmodel.html

TABLE IV: The prompt in the LLM: instructions are used in the system role.

Prompt on refining rules.
System role is as follows: **Task.** You are a {YARA|SemGrep} rule expert. Your task is to analyze and optimize the input rules. Please follow these steps to ensure the rules are complete and efficient: Analysis result: {user input} Rule: {user input} **Thought Process:** 1. Self-reflection: { ... ... } 2. Optimize Rules: { ... ... } **Output:** {YARA|SemGrep} rules
User's information is as follows: Input: {Analysis Result} refer to Section IV-A Input: {YARA|SemGrep rule} refer to Section IV-A

the thought process. The second output is a rule in either the YARA or Semgrep format. To guide the LLM, we leverage the few-shot learning technique by providing correct rule formats as references during rule generation. It is important to note that despite these measures, LLMs still introduce errors or hallucinations in the generated outputs.

B. Refining Rules

Two outputs from the previous stage (coarse-grained rules and analysis results) serve as the inputs for this task. The prompt (Table IV) provides the guidelines for this process. Specifically, the task is divided into 2 steps: rule analysis and rule optimization.

Self-reflection. We employ the self-reflection technique to guide the LLM in auditing coarse-grained rules. This approach leverages the LLM's capability to evaluate and critique its own prior outputs, enhancing overall performance [23, 24]. In the context of RULELLM, the self-reflection component ensures that the rules align with the analysis results. If discrepancies or inconsistencies are identified, the LLM revises the rules accordingly to maintain alignment and accuracy.

Rule Optimization. During this step, coarse-grained rules are optimized and merged into a single, scalable rule. This process ensures that the resulting rule is both effective and general enough to detect a broader range of malicious behaviors while maintaining efficiency. Several guidelines are followed in this subtask:

1) The `string` section should encapsulate malicious behaviors, such as API calls, file operations, and network activities. If this is not the case, the `string` section should be revised based on the analysis results.
2) Standard naming conventions are applied to the `string` section to enhance consistency and readability. When the `string` sections of multiple rules show similarities, regular expressions are used to manage potential variants.
3) Logical combinations (`all of them`, `any of them`, or regular expressions) are employed to merge rules. If two rules overlap in scope, the rule with smaller coverage is removed.
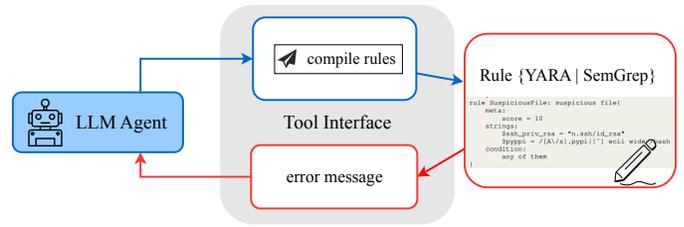

Fig. 4: Error correction module: an agent-based LLM fixes errors based on a tool interface.

4) The combined rule adheres to the required structure and format. A valid rule begins with the keyword `rule` followed by a unique identifier. Each YARA rule contains three sections: `meta`, `strings`, and `condition`.
5) The LLM minimizes resource-intensive operations (e.g., regular expressions) to optimize rule execution, ensuring efficiency without unnecessary overhead.

Output. This component outputs a calibrated rule in YARA or Semgrep format.

C. Aligning Rules

Due to the inherent limitations of LLM, errors or hallucinations in generated rules are inevitable. To address this, we propose an LLM-based agent designed to correct rule errors, as illustrated in Figure 4. Specifically, the agent utilizes the tool to compile the rules. A blue color indicates that the generated rule has passed the verification process, while red indicates that it has failed. If a rule fails, the agent refines it further by analyzing the error messages.

Tool. Although LLMs are capable of reading and generating text or images, they cannot interact with external environments or perform tasks such as searching or code execution. To overcome this limitation, we equip the agent with tools that enable communication between the LLM and the rule compiler. Specifically, we propose two manually created tools: one for compiling YARA rules and another for compiling Semgrep rules. The compiler does not raise an exception when a rule is correct; however, it will raise exceptions for issues such as syntax or compilation errors. These tools are implemented as a Python file, which can be stored and updated as needed.

Memory serves as the core of the agent, connecting tools, LLMs, and task execution. It is used to store short-term information during processing. Specifically, error messages generated by the rule compiler are treated as observations and stored in memory. The agent retrieves these error messages and feeds them into the task execution component. Each time a rule compilation fails, new error messages are generated. If a rule fails to compile multiple times, the memory history can grow excessively. To address this, the memory module retains only the two most recent compilation error messages.

Prompt. We design a prompt to guide the LLM-based agent in fixing rules, as detailed in Table V. The agent's tool generates error messages, which RULELLM uses to address and correct rule issues. RULELLM adheres strictly to the

TABLE V: The prompt in the LLM: 6 instructions are used in the system role; Red indicates error messages from the agent's observations.

Prompt on the rule fix.
Task. You are a {YARA|SemGrep} rule expert. Your task is to fix and optimize the input YARA rules. Please follow these steps to ensure the rules are complete, syntactically correct, and efficient: Error message: {error_info} Analysis result: {user input} Rule: {user input} **Instruction.** 1. Missing or Incomplete Parts: { ... ... } 2. Syntax Errors: { ... ... } 3. Undefined Strings in Conditions: { ... ... } 4. Regular Expression Issues: { ... ... } 5. Invalid 'meta' Field Values: { ... ... } 6. File Encoding Issues: { ... ... }
User's information is as follows: Input: {Analysis Result} Input: {YARA|SemGrep rule}
Agent's observation is as follows: Error: {error result}

TABLE VI: The details of the dataset for OSS malicious packages.

Category	Pkg. Num.	Deduplicated Num.	Avg. LoC
Malware	3,200	1,633	424
Legitimate	500	500	3,052

requirements and error messages to refine the rules, following these steps: (1) Ensure the rule contains all necessary components: 'meta', 'strings', and 'condition'. (2) Check for syntax issues such as unmatched brackets, unclosed quotes, and other errors. (3) Verify that all strings referenced in the condition section are properly defined in the 'strings' section. (4) Validate the correctness, efficiency, and expected matching behavior of all regex patterns. (5) Confirm that the meta fields are well-formatted and meaningful. (6) Ensure the rule is UTF-8 encoded without a BOM and that line endings align with the target environment. If a rule successfully compiles, RULELLM outputs the corrected rule. Otherwise, it attempts to fix the rule up to five times.

V. EVALUATION

In this section, we conducted a series of experiments to validate the effectiveness of RULELLM in generating rules for detecting malicious packages. Then, we provided a systematic analysis of those generated rules.

A. Experimental Setting

Implementation. We have implemented a prototype system of RULELLM via several open-source libraries. RULELLM's input is a malicious package, and the output is its corresponding YARA & Semgrep rules. RULELLM uses regular expression matching to extract the package metadata, where we use the re library [28] to implement the regex. RULELLM use the tokenize [29] library to convert source code to tokens,

TABLE VII: The details of baselines.

Category	Method
Existing Rules from SOTA Tools	Yara scanner [6], Semgrep scanner [5]
Score-based Approach	Prior works [31, 32]
Diverse LLMs	GPT-3.5 turbo, GPT-4o [17] Claude-3.5-Sonnet [19], Llama-3.1 70B [33]

and the CodeBERT embedding model to generate code's vectors. We leverage the Scikit-learn [27] library to implement the clustering algorithm to divide code fragments. RULELLM uses the LangChain [30] module to automatically generate rules based on the extracted metadata and code fragments.

Dataset. Table VI lists the details of the dataset, including 3,200 malware packages and 500 legitimate packages. The malware comes from the GuardDog [14] that provides PyPI malicious packages to the public via the GitHub repository. The legitimate packages come from the most popular PyPI packages (by download count) over one year [15]. We find there are many duplicate packages in the malware dataset, where their signatures are the same. After deduplicated, the number of malware is 1,633. The average number of malware code (LOC, Line of code) is 424 lines. For the legitimate packages, the average number of code segments is 3,052 LoC. It is obvious that the number of code segments in the malware packages is much smaller than that in the legitimate packages. The reason is straightforward: malware packages are designed for a specific purpose, such as data theft, backdoor access, or surveillance, and don't require extensive functionality.

Baselines. We compare RULELLM with several baselines: SOTA tools, automatic rule generation, and diverse LLMs. Table VII lists the details of the baselines.

(1) Existing Rules from SOTA Tools. The first category involves existing rules from YARA [6] and Semgrep scanners [5]. YARA scanner uses 4,574 YARA format rules to find risks and threats, covering vulnerabilities, malware, shells, mobile applications, emails, etc. Semgrep scanner uses 2,841 Semgrep format rules to find risks and threats, covering third-party software, cloud services, network communications, systems, network applications, etc. Those rules were written by developers, security professionals, or researchers. We use those rules to compare the effectiveness of rules generated by RULELLM.

(2) Score-based Approach. So far, there is no approach for generating rules to target OSS malware. Instead, several score-based approaches [31, 32] and tools [34] can generate signatures from binary files. Hence, we revise the score-based approach to adapt to OSS malware. (1) First, we use 3 types of scores to measure the importance of strings: isolation forest, information entropy, and TF-IDF (Term Frequency-Inverse Document Frequency). Each score is assigned a specific weight: isolation forest is given a weight of 1.2, TF-IDF has a weight of 1.0, and information entropy is weighted at 0.8. (2) Next, we apply a clustering algorithm (Section III-B) to partition both malware and legitimate packages (Table VI) into

TABLE VIII: Performance of RULELLM compared to baselines.

Rule Type	Accuracy	Precision	Recall	F1
RULELLM	81.4%	85.2%	91.8%	88.4%
Yara scanner	41.6%	35.0%	23.4%	28.0%
Semgrep scanner	56.2%	70.9%	32.0%	44.0%
Score-based	84.5%	47.8%	66.6%	55.7%

different code groups. (3) In each iteration, we pick up two groups (one from malware and one from legitimate) and use 3 scores to calculate the importance of strings between the two groups. (5) Strings with high scores (above a 0.9 threshold) are picked up into the 'string' part of the YARA format rule. The remaining parts of the rules are generated through a rule template.

(3) Diverse LLMs. We use several large language models (LLMs) to generate rules, including GPT-3.5 turbo, GPT-4o [17], Claude-3.5-Sonnet [19], Llama-3.1:70B [33]. The first three LLMs belong to online services, and Llama-3.1 belongs to the local LLM. We leverage APIs of online LLMs to generate rules. For Llama-3.1-70B, we deploy it on the local server. We inspect the diverse efforts of different LLMs on rule generation.

B. Performance

Effectiveness of Rules. First, we evaluate the effectiveness of rules generated by RULELLM for detecting malicious and legitimate packages. Table VIII lists the comparative performance of rules generated by RULELLM against various baselines, including Score-based, YARA scanner, and Semgrep scanner. We use 4 metrics to represent the performance, including accuracy, precision, recall, and F1 score. RULELLM achieved promising performance, with an accuracy of 81.4%, precision of 85.2%, recall of 91.8%, and F1 score of 88.4%, outperforming most other methods. Both scanners show significantly lower performance compared to RULELLM, with the Yara scanner performing the worst, achieving only 41.6% accuracy and an F1 score of 28.0%. The Score-based method performs well in accuracy (84.5%) but falls short on other metrics, indicating possible overfitting or reliance on specific criteria that may not generalize well. RULELLM demonstrates superior performance compared to the baselines, as evidenced by its high recall, precision, and F1 score. It effectively balances the identification of true positives and minimization of errors.

We further inspect the malware detection performance along with the number of matched rules. Figure 5 depicts the performance distribution (accuracy, precision, recall, and F1) of YARA rules, with the X-axis representing the number of matched rules. It is evident that when the matched rule number is equal to 1, the malware detection achieves the best performance. In addition, performance decreases continuously as the number of matched rules increases. This is because YARA-generated rules are highly specific and do not share similar patterns. Figure 6 shows the performance of Semgrep
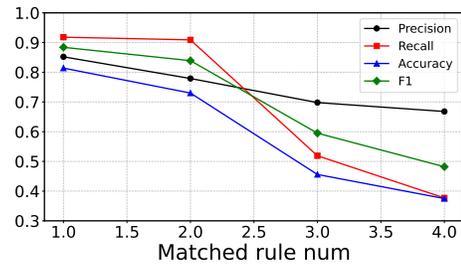

Fig. 5: YARA rule: the malware detection's performance along with the matched number.
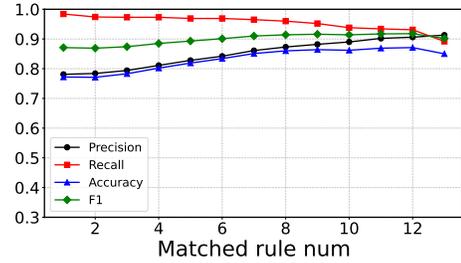

Fig. 6: Semgrep rule: the malware detection's performance along with the matched number.

rules relative to the number of matched rules. We observe that the performance of Semgrep-generated rules changes only slightly with the number of matched rules. When the number of matched Semgrep rules is equal to 9, malware detection reaches its peak performance. This can be attributed to the fact that Semgrep rules focus on code structures and static analysis, while YARA rules are centered on signatures and specific patterns. Hence, the Semgrep rules have broader patterns than the YARA rules. Overall, the number of matched rules (Figures 5 and 6) in malware detection demonstrates that RULELLM can effectively generate both YARA and Semgrep rules.

Table IX compares the performance of rules generated by various LLMs, including GPT-3.5-turbo, GPT-4o, Claude-3.5-Sonnet, Llama3.1:70b. GPT-4o achieves the highest performance across all metrics, with an accuracy of 81.4%, precision of 85.2%, recall of 91.8%, and an F1 score of 88.4%. Both Llama3.1:70b and GPT-3.5-turbo show moderate performance, with an accuracy of 72.6% and 74.5%. Notably, Claude-3.5-Sonnet's precision is lower at 75.0%, though its recall is relatively higher at 95.9%. These results suggest that GPT-4o excels at generating rules that are both precise and effective, capturing a high number of true positives and maintaining low false positives.

Malware Variant Detection. We further inspect whether rules generated by RULELLM can detect variants of OSS malware. We use the clustering algorithm (Section III-B) to divide malware packages into different groups. In each group, we use two malware packages to generate YARA rules, and the rest packages are unknown variants. We use those generated rules to detect unknown variants in the same group. The

TABLE IX: Performance of Rules Generated by different LLMs.

Rule Type	Accuracy	Precision	Recall	F1 Score
GPT-3.5 turbo	72.6%	78.4%	68.0%	72.8%
GPT-4o	81.4%	85.2%	91.8%	88.4%
Claude-3.5-Sonnet	75.0%	77.3%	95.9%	85.6%
Llama-3.1:70B	78.2%	68.0%	72.6%	77.4%

TABLE X: Ablation Experiment: impact of each component's effectiveness in RULELLM.

Approach	Precision	Recall
LLMs alone	62.9%	56.8%
LLM + Rule Alignment	79.2%	84.3%
LLM + Basic-unit Rule + Rule Alignment	81.90%	90.0%
LLM + Basic-unit Rule + Combination + Rule Alignment	85.2%	91.8%

overall detection rate is 90.32%, and the average detection rate is 96.62%. The results demonstrate that rules generated by RULELLM can detect potential variants.

Ablation Experiment. We validate the impact of each component in RULELLM (Figure 3): basic-unit rule creation, rule combination, and rule alignment. Specifically, we use 4 approaches: (1) LLMs alone; (2) LLM + rule alignment; (3) LLM + basic-unit rule + rule alignment; and (4) LLM + basic-unit rule + combination + rule alignment. To ensure a fair comparison, all prompts and their requirements (Table III-V) are consistently used in LLM. Table X lists each component's effectiveness in RULELLM. Directly using LLM to generate rules leads to a significantly low recall (56.8%) and relatively moderate precision (62.9%). Without the RULELLM's help, LLMs struggle with the rule generation task, missing a substantial portion of the rules. LLM + Rule Alignment can significantly improve both the precision and recall of generated rules. The reason is that the rule alignment can fix errors in rule formats, and this approach can find more useful rules. LLM + Basic-unit Rule + Rule Alignment yield a substantial increase in recall, jumping to 90.0%. The basic unit rules help this approach capture a broader range of true positives. RULELLM (a combination of various components) shows the highest performance across both precision and recall. Thus, RULELLM's effectiveness is not merely a result of using an advanced LLM; it stems from a combination of specialized techniques, optimizations, and a tailored agent-based architecture that improves rule quality.

C. In-depth Analysis

We provide an in-depth analysis of the rule quality, including the rule number, the precision per rule, and the coverage per rule.

Rule number. Table XI shows the rule number comparison between RULELLM and SOTA tools. RULELLM can generate two types of rules: Yara format and Semgrep format. We can see that RULELLM has 452 rules in the Yara rule format

TABLE XI: The rule number between RULELLM and SOTA tools.

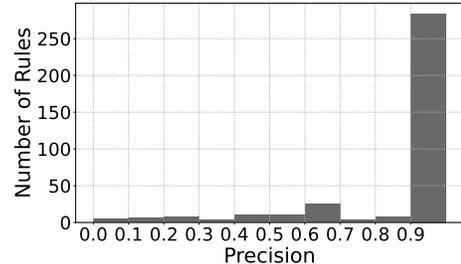| Category	SOTA Tool		RULELLM
	All Rules	OSS Malware	
Yara Rule Format	4,574	46	452
Semgrep Rule Format	2,841	334	311

Fig. 7: YARA rule's precision: the precision distribution for all rules generated by RULELLM.

and 311 rules in the Semgrep rule format. In comparison, SOTA tools have 4,574 total rules and 46 OSS malware rules in the Yara format (2,841 total rules and 334 malware rules in the Semgrep format). Rules from tools (Semgrep and Yara scanners) were written by security experts, requiring domain-specific knowledge and manual efforts. In addition, most of the YARA and Semgrep rules are not designed for OSS malware. Yara scanners are designed for signatures and are difficult to deal with malware deformation, and Semgrep scanners usually support taint analysis and string matching. Although the rule amount from RULELLM is smaller than that of SOTA tools, RULELLM shows the best performance (detailed in Table VII). Rules generated by RULELLM have broader detection coverage to recognize a variety of patterns, behaviors, or anomalies in malicious packages.

Precision per rule. We inspect the precision performance of every rule generated by RULELLM. Figure 7 depicts the distribution of 452 YARA rules' performance, where the X-axis is precision, and the Y-axis is the rule number. It is observed that 278 YARA rules have a high precision, nearly 98.2%, and the rest rules have various precisions. For each rule, the high precision indicates a high confidence in malware and threat detection. If a package matches a rule with high precision, it is confident that the package is malicious. Note that 65 YARA rules (452-387) do not match with any malicious packages.

Figure 8 depicts the distribution of 311 Semgrep rules' performance. Similarly, 158 Semgrep rules have a high precision, nearly 97.1%. We also find that nearly 40 Semgrep rules have close 0% precision and 62 Semgrep rules (311-249) do not match with any malicious packages. In terms of manual inspection, those rules use a highly specific taint-code structure, leading to poor performance.

Coverage per rule. Next, we use the number of detected
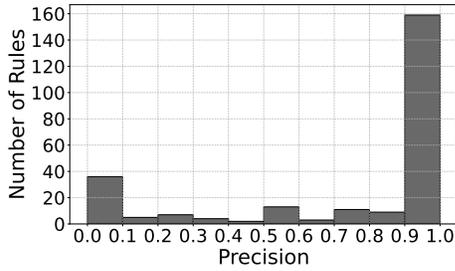
Fig. 8: Semgrep rule's precision: the precision distribution for all rules generated by RULELLM.
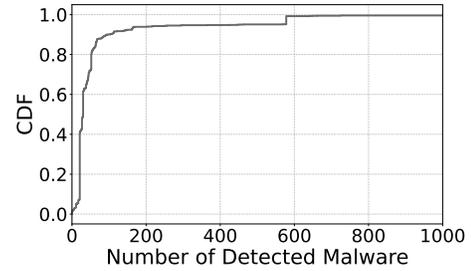

Fig. 10: Semgrep rule's coverage: the CDF of the detected malware per rule generated by RULELLM.

TABLE XII: YARA rules: detailed breakdown of rule categories and subcategories

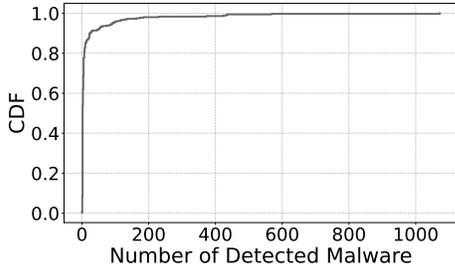| Category Name	Subcategory Name	Count
0. Metadata Related	Package Metadata Manipulation	92
	Version Number Deception	17
	Fake Dependency Metadata	18
	Author Information Spoofing	29
1. Malicious Behavior	Privilege Escalation	21
	Process Manipulation	25
	System Configuration Changes	70
	Persistence Mechanisms	87
2. Dependency Library	System Library Abuse	25
	Network Library Misuse	43
	Crypto Library Exploitation	7
	UI/Graphics Library Abuse	8
3. Setup Code	Malicious Setup Scripts	56
	Build Process Manipulation	11
	Installation Hook Abuse	39
	Configuration Tampering	28
4. Network Related	C2 Communication	66
	Data Exfiltration Channels	51
	Malicious Downloads	61
	DNS/Protocol Abuse	15
5. Obfuscation & Anti-Detection	Code Obfuscation	72
	Anti-Analysis Techniques	67
	Sandbox Evasion	9
	String/Pattern Hiding	35
6. Data Exfiltration	Credential Theft	8
	Environment Data Stealing	31
	Configuration File Extraction	2
	Sensitive Data Harvesting	53
7. Code Execution	Shell Command Execution	54
	Script Injection	29
	Process Creation	1
8. Application	Messaging Platform Abuse	35
	Social Media API Exploitation	2
	Cloud Service Misuse	18
	Development Tool Abuse	5
9. Malware Family	Known Trojan Families	12
	Backdoor Families	2
10. Other Rules	Unknown or Undetermined	13

Fig. 9: YARA rule's coverage: the CDF of the detected malware per rule generated by RULELLM.

malware packages to reflect the coverage of rules generated by RULELLM. A high number of detected malware packages indicates that a rule has broad coverage and identifies common patterns, while a low number of detected malware packages suggests that a rule is more specific and suited to a narrow set of patterns. Figure 9 shows the CDF of the detected malware number for YARA rules. It is evident that many YARA rules detect a small number of malware packages, whereas 80% rules cover fewer than 10 packages. However, 10 YARA rules detect over 100 malware packages, indicating broader and more common patterns. For example, a rule related to the fake version can detect 568 malware packages and a rule related to the C2 server can detect 185 malware packages. In short, most YARA rules use a specific string or regex, minimizing false positives by focusing on unique identifiers or signatures.

Figure 10 shows the CDF of detected malware number for Semgrep rules. It is observed that Semgrep rules have a broader range for detecting malware packages compared to YARA rules. Only 40% of Semgrep rules cover fewer than 10 malware packages, whereas other Semgrep rules with broader patterns may result in a higher false positive rate. In contrast, broad Semgrep rules have a high recall (detailed in Figure 6). Semgrep rules can be advantageous for identifying a broad spectrum of threats at the initial stage.

Overall, YARA rules differ from Semgrep rules in their matching patterns and targets. A broad rule is useful for a quick scan but risks false positives, whereas a specific rule excels in precision but may miss underlying threats.

D. Rule Analysis

We manually inspect the content of generated rules and categorize them into 11 categories and 38 subcategories. The categorization relies on the nature of the rules and profes-
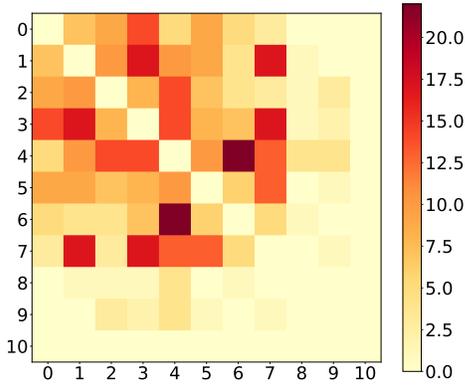
Fig. 11: Heatmap: the overlapping degree between different categories; the number indicates the rule category in Table XII.

sional experience. Table XII provides a detailed breakdown of various rule categories and their respective subcategories, along with the count of rules in each subcategory. It is evident that the category with the highest number of rules is "Malicious Behavior" with a total of 203 rules, followed closely by "Network Related" with 193 rules. "Malicious Behavior" encompasses actions that directly harm the system or escalate privileges. "Network Related" includes behaviors that involve network communication, such as command and control (C2) communication or data exfiltration. The high counts in categories underscore the complexity and prevalence of these threats from OSS malware. Conversely, low counts in certain categories (e.g., "Configuration File Extraction") may indicate less common or less complex threats. In short, our generated rules highlight the diverse range of rule categories and subcategories used to detect and mitigate various security threats.

(1) Non-exclusive category. The rule category and subcategory are not mutually exclusive, and a rule can belong to multiple categories and subcategories. The total number of YARA rules in Table XII is 1,217, compared to 452 YARA rules in Table XI. Figure 11 depicts the overlapping rules among different categories. For example, the 'Malware_Setuptools_PostHook' rule covers two categories: "Setup Code Rule' and "Network Related Rule". The reason is that RULELLM extracts rules from malicious packages, whereas a malware package may exhibit behaviors, e.g., a ransomware package might encrypt files while also exfiltrating data.

(2) Large Detection Range. Several rule categories have a large detection range: "Obfuscation & Anti-Detection", "Metadata-Related", "Code Execution", "Network Related", and "Data Exfiltration". Rules in those categories have a large range, where more than 1,000 packages are detected. In the "Code Execution" category, a rule detected 23.41 malware packages on average; in the "Obfuscation & Anti-Detection" category, a rule detected 19.50 malware packages on average. It concluded that some techniques(e.g., common code snippets, network behaviors, or obfuscation patterns) are generic and widely in OSS malware.

(3) Narrow Detection Range. Several rule categories have a narrow detection range: "Malware Family", and "Application". Rules in those categories have a small range, less than 5 packages per rule. The reason is that those malware packages are less well-known. Malware families are popular in conventional malware samples with different architectures and compilers, leading to binary polymorphism. However, OSS malware is a software package to organize artifacts and components. Malware families are not common in OSS malware. In brief, low-detection categories center around fewer but more specific matches of malware packages.

VI. Discussion & Limitations

Data Leakage [35, 36] leads to the concern of the RULELLM's performance. If the LLM uses the GuardDog dataset to pre-train the model with the same target, the validity of the performance in RULELLM may be overstated. There are 2 manners to mitigate the impact of data leakage of LLMs. First, we can use malicious packages whose release time is newer than the cutoff date of model pre-training. In our experiments, there are 78% malicious packages whose release time is newer than Dec./2023, and GPT-4o's cutoff date is Oct./2023. There is a high probability that those malicious packages do not have data leakage issues. Second, even though LLM uses the GuardDog dataset to pre-train the model, the LLM still struggles with the rule generation task. Our experimental results (Table X) show that directly using LLM to generate rules leads to a significantly low recall (56.8%) and relatively moderate precision (62.9%).

LLMs' Limitations. RULELLM's performance mainly relies on the reasoning ability of LLMs. However, as LLMs are trained on general datasets when handling specialized knowledge such as malicious packages, inaccuracies are inevitable. This can result in the generated rules being overly broad, leading to a high number of false positives, or only capable of identifying particular samples. In our RULELLM, the LLM can achieve 85.2% precison. Another problem is the hallucinations caused by LLMs, where some rules may be fabricated content, or confuse truth and falsehood. The reason for fabricated rules is that the LLM lacks a definitive ground truth for unknown malicious packages and unseen risks. Due to context length limitations, LLMs struggle to handle excessively long malicious code, such as obfuscated code or payload with base64 encoding.

Retrieval Augmented Generation (RAG) can provide external knowledge (e.g., databases) to LLMs for improving performance and mitigating hallucinations. RULELLM belongs to a knowledge-intensive domain, where RAG can update security knowledge to guarantee the generated rule quality. This work only integrates prompt engineering (task decomposition, CoT, reflection, and few shots) into RULELLM without RAG. Note that prompt engineering and RAG are not mutually exclusive and can complement each other to improve LLMs' capabilities.

Fine-Tuning is to train a pre-trained LLM on a domain-specific dataset. RULELLM can achieve a better performance

when we tailor an LLM to the rule generation task. Specifically, there are three requirements for fine-tuning LLM: (1) a pre-trained model (e.g., Llama 3.1), (2) a labeled domain dataset; and (3) training the LLM with Transformers. However, we lack a labeled domain dataset for fine-tuning LLM. The labeled data should be in the supervised format, denoted as *(a malicious package, a rule)*.

VII. RELATED WORK

Software Supply Chain Attack. Nowadays, OSS ecosystems have millions of packages [37, 38, 39], and the package dependencies are becoming very complex [40, 41, 42, 43]. Malicious packages [44, 45, 46, 47, 48, 45] often contain code specifically crafted to perform unwanted or harmful actions on a system, including obfuscated code, data exfiltration, self-execution, typosquatting, dependency confusion, and backdoors. Pfretzschner and ben Othmane [49] proposed a detection algorithm for dependency-based attacks on Node.js, and Staicu et al. [50] proposed a deep understanding of the injection attack on Node.js. Ladisa et al. [51] proposed a general taxonomy of conceptual attack vectors in the OSS ecosystem. Guo et al. [45] collected malicious packages in PyPI ecosystems, leveraged the case study to analyze the malicious behaviors.

Vulnerable packages [52, 53] refer to software libraries or components that contain security flaws, misconfigurations, or weaknesses that can be exploited by malicious actors. Alfadel et al. [54] studied a collection of 550 vulnerabilities affecting 252 PyPi packages, and their analysis showed that vulnerabilities grew over time, and the most common was XSS vulnerabilities. Ponta et al. [55] proposed a code-centric scheme for detecting, analyzing, and mitigating vulnerabilities in software packages. Woo et al. [56] traced a reported vulnerability back to its origin in the codebase, and combined static analysis, metadata extraction, and historical codebase tracking to identify the original vulnerability.

OSS ecosystems leak sensitive data and compromise users' privacy, caused by flaws, insecure configurations, and improper use of third-party dependencies. Vaidya et al. [57] pointed out private information is leaked in the code of software packages, including key files and API keys embedded in the code. Xiao et al. [58] proposed an attack that abuses hidden attributes, which attackers can exploit to obtain confidential data, bypass security checks, and launch denial-of-service attacks.

Large Language Model is primarily based on the Transformer architecture with extensive pre-training on large-scale training data, and it has demonstrated remarkable advances across various domains. Previous works [21, 22, 59] proposed a planner to extend LLMs' capabilities for dealing with complex tasks, such as generalization and reasoning using task decomposition techniques such as Chain-of-thought [21] and Tree-of-thought [22]. Yao et al. [59] proposed a general strategy of self-reflection (called ReAct) based on the feedback of LLMs to improve their reasoning skills. Similarly, Reflexion [23] and Chain of Hindsight [24] use human feedback

(e.g., errors) to fine-tune LLMs. Another distinct approach is to use external resources to build an autonomous agent, enabling LLMs to interact with the environment (tools or APIs). Wang et al. [60] proposed an embodied lifelong learning agent based on LLMs. Huang et al. [61] let an LLM self-improve its reasoning without supervised data by asking the LLM to lay out different possible results.

Meanwhile, LLMs can be applied in the software engineering domain, such as OpenAI Codex [18] and GitHub Copilot [62]. Many prior works leveraged LLMs to resolve specific tasks in the software engineering domain [63, 64, 65, 66, 67, 68]. Ma et al. [63] and Sun et al. [64] explored the capabilities of LLMs when performing various program analysis tasks, including control flow graph construction, call graph analysis, and code summarization. There are many prior works using LLMs to resolve specific tasks in the security domain [69, 70]. Pearce et al. [13] proposed to leverage LLM to help security professionals reverse engineer the binary application for automatically repairing vulnerabilities. Li et al. [11] presented LLM capabilities in the static analysis of finding vulnerabilities. Feng and Chen [71] proposed to use LLM to replay Android bugs automatedly. Pearce et al. [9] examined zero-shot vulnerability repair using LLMs.

Security Rules. Security detection tools are widely used in the software engineering domain to detect security vulnerabilities, malicious packages, and privacy leaks. YARA-scanner [6, 31, 32] is a tool for leveraging YARA rules to analyze the malicious features in the software package. AppInspector [7] includes 16 aspects and a total of 712 rules, which are used to check regular expression-based malicious patterns (e.g., reverse shell) in the text of each file in the package. Semgrep [5] is a tool for pattern matching and searching in structured data, which can be used to detect sensitive information leakage, configuration errors, etc.

VIII. CONCLUSION

In this paper, we demonstrate that LLMs can be an effective tool for generating rules in OSS ecosystems. RULELLM leverages LLMs to systematically analyze metadata and code, creating accurate YARA & Semgrep rules without manual efforts, achieving a precision of 85.2% and a recall of 91.8%. Through its prototype implementation, RULELLM has proven its effectiveness and adaptability, showcasing its capability to enhance current security practices and outperform established tools. Our work holds promise for future advancements in OSS security, particularly as a scalable solution to evolving SSC threats.

REFERENCES

[1] Sonatype. (2021) State of the software supply chain. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021.

[2] E.Roth. (2021) Open source developer corrupts widely-used libraries, affecting tons of projects. https://www.theverge.com/2022/1/9/22874949/developer-corrupts-open-source-libraries-projects-affected.

[3] I. Pashchenko, D.-L. Vu, and F. Massacci, "Preliminary findings on foss dependencies and security," 2020.

[4] C. org. (2022) Apache Log4j Vulnerability. https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.

[5] semgrep org. (2019) SemGrep rules for the security static analysis. https://github.com/semgrep/semgrep.

[6] N. Naik, P. Jenkins, R. Cooke, J. Gillett, and Y. Jin, "Evaluating automatically generated yara rules and enhancing their effectiveness," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2020, pp. 1146–1153.

[7] M. org. (2023, accessible) The tool identifies coding features of first or third party software components. https://github.com/microsoft/ApplicationInspector.

[8] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," *arXiv preprint arXiv:2312.12575*, 2023.

[9] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2339–2356.

[10] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen, and Q. Liao, "A repository-level dataset for detecting, classifying and repairing software vulnerabilities," *arXiv preprint arXiv:2401.13169*, 2024.

[11] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "The hitchhiker's guide to program analysis: A journey with large language models," *arXiv preprint arXiv:2308.00245*, 2023.

[12] X. Shang, S. Cheng, G. Chen, Y. Zhang, L. Hu, X. Yu, G. Li, W. Zhang, and N. Yu, "How far have we gone in stripped binary code understanding using large language models," *arXiv preprint arXiv:2404.09836*, 2024.

[13] H. Pearce, B. Tan, P. Krishnamurthy, F. Khorrami, R. Karri, and B. Dolan-Gavitt, "Pop quiz! can a large language model help with reverse engineering?" *arXiv preprint arXiv:2202.01142*, 2022.

[14] E. Wang. (2020) The CLI tool that allows to identify malicious PyPI and npm packages. https://github.com/DataDog/guarddog.

[15] H. van Kemenade. (2024) Top PyPI Packages. https://hugovk.github.io/top-pypi-packages/.

[16] X. Zhang, "Malware detection rule generator." https://github.com/zhang-xr/RuleLLM, 2024.

[17] O. Org. (2023) The OpenAI API is used for a range of models and fine-tune custom models. https://platform.openai.com/docs/introduction.

[18] ——. (2023) OpenAI Codex: AI system that translates natural language to code. https://openai.com/blog/openai-codex.

[19] A. Org. (2023) Claude is a next generation AI assistant. https://claude.ai/.

[20] G. Org. (2023) Google's Gemini family for the multi-modal model. https://poe.com/Gemini-Pro.

[21] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[22] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.

[23] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[24] H. Liu, C. Sferrazza, and P. Abbeel, "Languages are rewards: Hindsight finetuning using human feedback," *arXiv preprint arXiv:2302.02676*, 2023.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pretrained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[26] Numpy. (2009) numpy: Randomly permute a sequence. [Online]. Available: http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.permutation.html

[27] Scikit-learn. (2007) Machine learning library for the python language. http://scikit-learn.org/stable/index.html.

[28] F. Jeffrey. (2009) Regular expression operations. https://docs.python.org/3/library/re.html.

[29] NLTK. (2001) A suite of libraries and programs for symbolic and statistical natural language processing. http://www.nltk.org/.

[30] LangSmith. (2023) LangChain, a unified platform for debugging, testing, evaluating, and monitoring your LLM applications. [Online]. Available: https://blog.langchain.dev/announcing-langsmith/

[31] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt, "Automatic yara rule generation using biclustering," in *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, 2020, pp. 71–82.

[32] M. Brengel and C. Rossow, "{YARIX}: Scalable {YARA-based} malware intelligence," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3541–3558.

[33] M. Org. (2023) Llama 2: open source, free for research and commercial use. https://llama.meta.com/llama2/.

[34] A. Org. (2023) Vovk — Advanced Yara rule generator. https://github.com/malienist/vovk?tab=readme-ov-file.

[35] C. Xu, S. Guan, D. Greene, and M.-T. Kechadi, "Benchmark data contamination of large language models: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2406.04244

[36] W. Shi, A. Ajith, M. Xia, Y. Huang, D. Liu, T. Blevins, D. Chen, and L. Zettlemoyer, "Detecting pretraining data from large language models," *arXiv preprint arXiv:2310.16789*, 2023.

[37] E. Constantinou and T. Mens, "An empirical comparison of developer retention in the rubygems and npm software ecosystems," *Innovations in Systems and Software Engineering*, vol. 13, no. 2, pp. 101–115, 2017.

[38] Y. Ma, "Constructing supply chains in open source software," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 458–459.

[39] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "World of code: an infrastructure for mining the universe of open source vcs data," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 143–154.

[40] A. Serebrenik and T. Mens, "Challenges in software ecosystems research," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, 2015, pp. 1–6.

[41] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[42] N. Zahan, L. Williams, T. Zimmermann, P. Godefroid, B. Murphy, and C. Maddila, "What are weak links in the npm supply chain?" *arXiv preprint arXiv:2112.10165*, 2021.

[43] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1513–1531.

[44] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio,

and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," *arXiv preprint arXiv:2002.01139*, 2020.

[45] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in pypi ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 166–177.

[46] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 23–43.

[47] E. Wyss, A. Wittman, D. Davidson, and L. De Carli, "Wolf at the door: Preventing install-time attacks in npm with latch," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1139 – 1153. [Online]. Available: https://doi.org/10.1145/3488932.3523262

[48] P. Ladisa, H. Plate, M. Martinez, O. Barais, and S. E. Ponta, "Towards the detection of malicious java packages," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 63 – 72. [Online]. Available: https://doi.org/10.1145/3560835.3564548

[49] B. Pfretzschner and L. ben Othmane, "Identification of dependency-based attacks on node. js," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–6.

[50] C.-A. Staicu, M. Pradel, and B. Livshits, "Understanding and automatically preventing injection attacks on node. js," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[51] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1509–1526.

[52] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

[53] ——, "On the evolution of technical lag in the npm package dependency network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 404–414.

[54] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 446–457.

[55] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[56] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "{V0Finder}: Discovering the correct origin of publicly reported software vulnerabilities," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3041–3058.

[57] R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security issues in language-based sofware ecosystems," *arXiv preprint arXiv:1903.02613*, 2019.

[58] F. Xiao, J. Huang, Y. Xiong, G. Yang, H. Hu, G. Gu, and W. Lee, "Abusing hidden properties to attack the node. js ecosystem," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2951–2968.

[59] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[60] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.

[61] J. Huang, S. S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han, "Large language models can self-improve," *arXiv preprint arXiv:2210.11610*, 2022.

[62] M. Org. (2023) Copilot: The AI developer tool. https://github.com/features/copilot.

[63] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu, "The scope of chatgpt in software engineering: A thorough investigation," *arXiv preprint arXiv:2305.12138*, 2023.

[64] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[65] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.

[66] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for 0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[67] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[68] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[69] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[70] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Can large language models identify and reason about security vulnerabilities? not yet," *arXiv preprint arXiv:2312.12575*, 2023.

[71] S. Feng and C. Chen, "Prompting is all your need: Automated android bug replay with large language models," *arXiv preprint arXiv:2306.01987*, 2023.