Fact-Aligned and Template-Constrained Static Analyzer Rule Enhancement with LLMs

Zongze Jiang*†‡, Ming Wen*†‡‖, Ge Wen‡, Hai Jin*¶
*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security
‡School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
¶Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China
Email: {jiangzongze, mwenaa, gwen, hjin}@hust.edu.cn

***Abstract*—Static analyzers are vital to ensure software quality, but often produce false alarms. In this paper, we focus on the challenging task, directly refining defective static detection rules in the analyzer with Large Language Models to mitigate false positives/negatives fundamentally. This paper introduces RULEREFINER, a novel multi-stage framework for static analyzer rule refinement. Specifically, RULEREFINER systematically employs LLMs by integrating dynamic profiling information for fact-based rule-code alignment, performing differential fault localization to accurately pinpoint error sources, and utilizing targeted templates to guide and constrain LLM-based modifications for precise and minimally disruptive enhancements. Evaluated on 218 real-world refinement tasks, RULEREFINER achieved a pass@5 score of 80.28%, significantly outperforming all selected LLM-based baselines under the same settings. Moreover, the rules refined by RULEREFINER demonstrated high generalization capability comparable to those written by human experts.**

***Index Terms*—rule-based static analysis, software fault localization, software refinement**

I. INTRODUCTION

Maintaining high code quality is fundamental to ensuring the reliability, security, and long-term maintainability of software systems. Static analysis tools have emerged as indispensable components of modern development workflows, providing automated mechanisms for detecting code defects, security vulnerabilities, and violations of coding standards or specifications. By enabling early detection of such issues during the development lifecycle, these tools significantly reduce the cost and complexity of downstream debugging and maintenance efforts [1]–[10]. Prominent examples include SonarQube [11], ErrorProne [12], and Infer [13]. These tools not only facilitate continuous quality assurance but also support the development of robust and secure software at scale.

In addition to these off-the-shelf tools, several widely adopted static analysis frameworks now provide user-friendly interfaces that enable developers and maintainers to define custom detection rules tailored to their specific application domains such as Semgrep [14], CodeQL [15] and PMD [16]. These frameworks significantly enhance the adaptability of static analysis and the detection of project-specific coding issues and security vulnerabilities. Fig. 1 presents a simplified Semgrep rule aimed at identifying *operating system command injection* (OSCI) vulnerabilities in Java code, demonstrating the practical utility of such static analysis frameworks.

‖ Corresponding author. Ming Wen is also affiliated with Wuhan JinYinHu Laboratory.

```
1  // Java Code with OSCI problems.
2  public class Test {
3    public void bad1(String userData){
4      ProcessBuilder pb = new ProcessBuilder();
5      pb.command("sh", "/c", userData);
6    }
7    public void bad2(ProcessBuilder builder,
         CustomClass userData){
8      String input = userData;
9      builder.command("bash", input);
10   }
11   public void bad_fn(String userData){
12     ProcessBuilder pb = new ProcessBuilder();
13     String cmd = trim(userData);
14     String shell = "sh"
15     String arg = "/c"
16     pb.command(shell, arg, cmd);
17   }
18 }
```

```
19 # Semgrep rule pattern for OSCI problem
20 patterns:
21   - pattern-inside: |
22     $FUNCDECL(..., $USER_DATA, ...){...}
23   - pattern-either:
24     - pattern: |
25       $PB.command(..., $USER_DATA, ...)
26     - patterns:
27 +     - pattern-either:
28           - pattern-inside: |
29             $X = $USER_DATA; ...
30 +         - pattern-inside: |
31 +           $X = trim($USER_DATA); ...
32       - pattern: $PB.command(..., $X, ...);
```

Fig. 1: A simplified Semgrep detection rule (L_{19}-L_{32}) to detect OSCI bugs and three Java defective code examples (L_1-L_{18})

Despite their practical utility, static detection rules are inherently imperfect [17]–[22]. In practice, when developers design such rules, they typically begin by examining the common syntactic and semantic characteristics of known buggy code examples [23]. These observations are then abstracted

into generalized and approximate rules intended to capture similar patterns of potential vulnerabilities or defects in unseen code. However, due to the inherent complexity of software systems and the expressive richness of programming language grammars, it is infeasible to account for all possible variations and corner cases. As a result, static analysis tools will often generate plenty of false positives or negatives in real-world applications [17]–[19], [24].

Over recent years, many techniques have been developed to mitigate false alarms for static analyzers [25]–[30]. For example, BayeSmith [31] filters and prioritizes alarms by a learning-based approach. FuzzSlice [32] generates and fuzzes function-level code slices to prune false positives. More recently, *Large Language Models* (LLMs) have demonstrated promising perforamnce in various software engineering tasks leveraging their strong code comprehension capabilities [33]–[38]. Therefore, it has also been leveraged to refine static analysis outputs by re-evaluating warnings to discern true positives [39], [40]. However, these post-processing strategies do not address the fundamental issue of generating false negatives [41] and can further incur significant costs due to repeated LLM invocations for each violation as reported by the analyzer. In this paper, we focus on ***directly refining existing static detection rules to mitigate false positives or negatives fundamentally, rather than performing post-mortem analysis to filter false positives.***

Technical Challenges. However, refining detection rules to fix known false positives/negatives (i.e., denoted as ***defect-revealing cases*** in this study) is non-trivial. LLMs, even equipped with advanced prompt strategies such as few-shot and chain-of-thoughts, still perform poorly.

Challenge 1: Static detection rules are typically expressed in formats distinct from the target code, posing a semantic gap that is challenging for LLMs to bridge. Static analysis engines are often complex, featuring custom rule grammars, intermediate representations, and unpredictable optimizations, making it difficult for LLMs to align detection rules with target code snippets. For example, Fig. 1 shows a rule from Semgrep and several target code snippets. Despite the distinct representations, the pattern at L_{22} actually matches all three code snippets. However, lacking domain knowledge of analyzer engine implementations, LLMs often struggle to bridge the semantic gap between detection rules and the target code.

Challenge 2: A static detection rule is often complex, capturing multiple variants of similar issues, making it non-trivial to localize the root cause for a given false positive or negative. Violation cases associated with a given rule can exhibit significant variability. While there are common code features shared across all variants, each variant may also possess relatively specific characteristics. For example, in Fig. 1, the pattern observed at line L_{22} is shared across all three code snippets, whereas the patterns at lines L_{29} and L_{31} are more specific to individual cases. The presence of diverse code variants often necessitates the creation of complex detection rules, in which multiple intertwined logic components are embedded within a single rule. As a result, when such a rule yields a false positive or false negative in a real-world code example, the entangled nature of the logic significantly hinders LLMs from accurate root cause analysis.

Challenge 3: LLMs struggle to refine static detection rules without disrupting their carefully designed logic or introducing new false positives and negatives. If not properly constrained, LLMs may compromise the carefully designed logic of original detection rules, inadvertently impairing their ability to correctly identify true positives or filter out true negatives. Such modifications often introduce new false positives or false negatives (i.e., regressions), ultimately degrading the accuracy and reliability of the refined rules. This risk is heightened by the lack of semantic understanding of the rule's intent and its interaction with complex code patterns.

Our Solution. To address the above limitations, we propose RULEREFINER, a framework featuring *fact-based alignment*, *differential fault localization*, and *template-guided refinement*, mainly based on the following key insights:

Insight 1: Leverage the static analysis engine to align the logic between detection rules and the code under analysis. Specifically, we utilize dynamic profiling information generated by static analysis engines, rather than relying solely on the opaque reasoning of LLMs. This approach provides a concrete, empirical basis for aligning detection rules with the target code, effectively bridging the semantic gap caused by the lack of detailed domain knowledge about the working mechanisms of the analysis engine.

Insight 2: Perform differential analysis between the correctly handled regression cases and the defect-revealing case to localize the root cause. We first abstract the detection rule into a graph representation, where each path represents a concrete detection logic (see Section III-C1). We then locate the root cause by comparing the different detection paths of the defect-revealing case and the correctly handled cases. In particular, we focus on the correctly handled cases that have similar detection paths, but also have subtle differences from the defect-revealing ones, to reveal the potential fault locations.

Insight 3: Design targeted templates to constrain and guide LLMs to generate localized and precise modifications. Instead of querying LLMs to refine detection rules in an unconstrained, end-to-end manner, we design templates tailored to the specific root causes [42]–[49]. These templates serve to guide and constrain the inference of LLMs, ensuring that any enhancements to the static detection rules are targeted and minimally disruptive. This structured approach helps preserve the original intent and logic of the rule, while allowing for precise and constrained enhancement.

We implemented a prototype of RULEREFINER and evaluated its refinement capabilities on 218 real-world Semgrep rule issues. The comprehensive evaluation demonstrated the effectiveness of RULEREFINER, achieving up to an 80.28% pass@5 success rate of refinement based on the DeepSeek model. RULEREFINER significantly outperformed LLM-based baseline approaches by 24%-68% relatively, underscoring the effectiveness of our novel techniques. Our ablation study confirm that both its differential fault localization and template-

based refinement components are crucial, each substantially contributing to the overall performance. Furthermore, we leverage the existing analyzer testing tool [50] to generate over 200K variants to test those refined rules by RULEREFINER systematically. The result shows that rules refined by RULEREFINER demonstrated capabilities on par with the expert-written rules in detecting defects across other code variants, while maintaining high precision on regression cases.

Our paper makes the following main contributions:

- ***Originality:*** To the best of our knowledge, we are the first to leverage LLMs to address the challenging task, refinement of the off-the-shelf real-world static analyzer rules. This brings new opportunities for research in automated rule refinement and offers novel insights.
- ***Approach:*** We propose RULEREFINER, a novel multi-stage framework designed to overcome the limitations of end-to-end LLM approaches. RULEREFINER incorporates fact-aligned, template-constrained novel designs to achieve precise and effective refinement.
- ***Evaluation:*** We evaluate RULEREFINER on 218 real-world rule issues sourced. The result demonstrates that RULEREFINER effectively addresses rule refinement tasks and significantly outperforms the baseline methods.
- ***Open Source:*** We have open-sourced our dataset, the implementation of RULEREFINER, and evaluation scripts at: https://github.com/CGCL-codes/RuleRefiner/ for reproducibility and further research.

II. BACKGROUND

A. Static Analyzers and Detection Rules

Static code analysis tools examine source code for issues like vulnerabilities and errors without execution. They typically contain two parts: an *analysis engine* and a set of *detection rules*. The engine performs general analysis (e.g., creating Abstract Syntax Trees or Control Flow Graphs) and provides underlying capabilities for rules such as taint analysis. A specific rule is designed to identify a particular problem (e.g., hard-coded passwords, SQL injection), which is usually built based on *predicates* and *logical operations*. Predicates pinpoint specific code features (e.g., Semgrep's `pattern`, PMD's `XPath`, CodeQL's `predicate`). Logical operators (AND, OR, NOT) then combine predicates into complex rules.

B. Explanation of A Motivation Example

Fig. 1 showcases a Semgrep rule for command injection detection. Statements under tags like `pattern` or `pattern-inside` serve as predicates (e.g., at L_{24}, L_{28}). In these statements, "`$X`" matches variables that are shared between predicates, and "..." acts as a wildcard. For instance, the predicate at L_{25} detects calls to `command` with one argument from parameters of the caller (L_{22}).

These predicates are then combined using logical operations, which are indicated by specific tags in the rule structure. Tags like `patterns` (see L_{26}) imply that the enclosed predicates are combined with the AND logic. Similarly, `pattern-either` (see L_{23}) signifies an OR combination, and `pattern-not` indicates the NOT logic. Therefore, the overall detection logic of this example is as follows: the rule identifies code instances where any parameter of a function (as defined by the predicate at L_{22}) is found to flow into an argument of a call to the `command` function (the sink, identified by predicates like those at L_{25} or L_{32}). This flow can be direct (e.g., captured by the logic associated with L_{25}) or indirect (e.g., traced through an assignment statement or certain function call specified by predicates at L_{29} and L_{31}).

III. APPROACH

A. Problem Formulation

1) Rule Abstraction: Real-world detection rules are complex and diverse. For simplicity and generality, we first abstract static detection rules. Based on the formulation of existing works [23], [51], [52] and our observations of the real-world detection rules, we represent a detection rule as a set of *predicates* together with *logical operators*. Formally, a *predicate* can be defined as follows:

Definition 1. *(Predicate). A predicate takes the code under detection as input and returns the predicate satisfaction status:*

$$p : code \rightarrow \texttt{True} \mid \texttt{False}$$

where "code" represents the code under detection, and the returned boolean value indicates whether the predicate is satisfied (i.e., *satisfaction status*).

Definition 2. *(Detection Rule). A detection rule is combined by a series of predicates with logical operators. A compound detection rule r can be defined recursively:*

$$r := p \mid r \wedge r \mid r \vee r \mid \neg r$$

Based on such abstraction, the rule in Fig. 1 can be denoted as $r_{osci} : p_{22} \wedge (p_{25} \vee ((p_{29} \vee p_{31}) \wedge p_{32}))$, where the subscripts denotes the line numbers where the predicate is located.

2) Test and Validation: As mentioned in Section I, there are no rules with perfect precision and recall in practice. In this paper, we use both the ***regression cases*** (i.e., true positive and true negative cases) and the ***defect-revealing cases*** (i.e., false positive and false negative cases) as the test suite (i.e., denoted as T) for the refinement task.

Definition 3. *(Test Cases). Formally, a test case is composed of the code under detection and the expected detection results, which is denoted as t,*

$$t :< t_{code}, t_{expected} >, t_{expected} \in \{true, false\}$$

where t_{code} is the code under detection, $t_{expected}$ indicates the expected detection result. In particular, $t_{expected}$ is true if t_{code} is expected to be reported as a violation of the bug encrypted by r, i.e., positive; otherwise negative.

Definition 4. *(Defective Rule). If r is defective, there is at least one test case t:*

$$\exists t \in T, t_{expected} \neq r(t_{code})$$

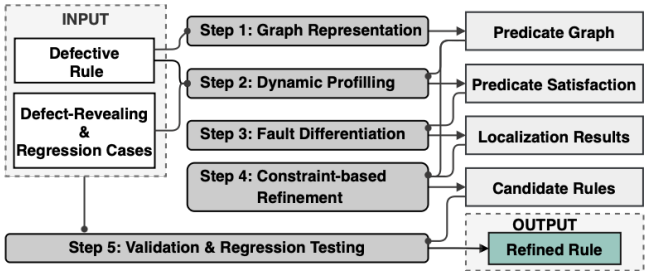
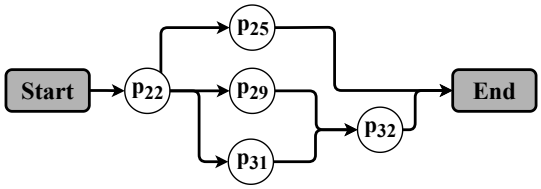


Fig. 2: Overview of RULEREFINER. Dot-ended arrows denote input streams, triangular-ended arrows represent output streams.



$\mathbf{r_{osci}}$: $p_{22} \wedge (p_{25} \vee ((p_{29} \vee p_{31}) \wedge p_{32}))$

Fig. 3: The predicate graph for the semgrep rule in Fig. 1.

where r(t_{code}) refers to the actual detection result of code ϵ_{code} against rule r under the static analuzer engine.

There can be multiple defects together with multiple defect-revealing cases in one rule. It is noted that we only focus on a single defect-revealing case for refinement each time.

Definition 5. *(Defect-Revealing and Regression Cases). Formally, we denote the defect-revealing case t in Definition 4 as ϵ, while the regression cases in T as T_r:*

$$T = \{\epsilon\} \cup T_r, \epsilon_{expected} \neq r(\epsilon_{code}), \forall t \in T_r, t_{expected} = r(t_{code})$$

3) Rule Refinement: Finally, the formulation of our rule refinement task is defined as follows:

Definition 6. *(Rule Refinement). The refinement task is denoted as a function F:*

$$F : (r, \epsilon, T_r) \rightarrow \hat{r}, \epsilon_{expected} \neq r(\epsilon_{code}),$$
$$\forall t \in \{\epsilon\} \cup T_r, t_{expected} = \hat{r}(t_{code})$$

where r denotes the defective detection rule, ϵ refers to the defect-revealing case, T_r refers to the regression cases, and $\hat{r}$ is the successful refined rule.

B. Overview

The high-level architecture of RULEREFINER is depicted in Fig. 2. RULEREFINER takes an original defective detection rule r, a specific defect-revealing test case ϵ, and a set of regression test cases T_r as inputs. The output is a refined detection rule $\hat{r}$, which can pass the defect-revealing case ϵ while preserving the original logic by successfully passing all regression test cases in T_r. RULEREFINER contains five principal stages:

- **Graph Representation:** The defective rule r is initially translated into a *predicate graph* in which distinct paths in the graph correspond to different branches of detection logic embedded in the rule.
- **Dynamic Profiling:** This process profiles the satisfaction status of each predicate, mapping the code under analysis to specific detection logic paths in the predicate graph.
- **Differential Fault Localization:** RULEREFINER then performs differential analysis based on predicate graphs and the profiled satisfaction facts, comparing the defect-revealing case ϵ with regression cases T_r. By focusing on the difference among them, the analysis isolates one or a few potentially **faulty predicates as the root cause**.
- **Template-Constrained Refinement:** For each localized fault, RULEREFINER generates a corresponding refinement template and then prompts LLMs to make targeted modifications to the defective rule based on the template.
- **Validation:** Finally, the refined rule $\hat{r}$ is rigorously validated against both the defect-revealing test case ϵ and all regression test cases T_r, ensuring that the refinement has rectified the given defect without compromising the original correct detection logic.

C. Graph Representation and Dynamic Profiling

1) Prediacte Graph: To simplify the representation and analysis of abstracted rules, we translate each rule into a graph structure termed a *predicate graph*.

Definition 7. *(Predicate Graph). A* predicate graph *is a directed acyclic graph (DAG):*

$$G = \langle s, e, N, E \rangle$$

where s is the unique entry node, e is the unique exit node, N is the set of all nodes, and E is the set of directed edges. The set of all paths $\mathcal{P}$ in G from s to e is defined as:

$$\mathcal{P} = \{\pi = (v_1, v_2, \ldots, v_m) \mid v_1 = s, v_m = e, \text{ and } (v_k, v_{k+1}) \in E \text{ for all } 1 \leq k < m\}.$$

In a graph, each node $v \in N$ (excluding s and e) typically corresponds to an atomic predicate within the rule formulation. The edges E represent the logical relationships governing these predicates. Specifically, the logical AND operator is realized by a sequential conjunction of nodes, while the logical OR operator is represented by parallel disjunction branches. Note that for the AND operator, the graph enforces a specific order: the left-hand predicate is always placed closer to the entry node s than the right-hand one, thus ensuring there is only one unique graph for each rule.

Fig. 3 shows the predicate graph for the abstracted rule $(p_{22} \wedge (p_{25} \vee ((p_{29} \vee p_{31}) \wedge p_{32})))$. There are three paths from the entry to the end, which correspond to the three conjunctive clauses in the disjunctive normal form of the rule (i.e., $(p_{22} \wedge p_{25}) \vee (p_{22} \wedge p_{29} \wedge p_{32}) \vee (p_{22} \wedge p_{31} \wedge p_{32})$). The rule is satisfied if any one of its conjunctive clauses (e.g., $(p_{22} \wedge p_{25})$) is satisfied.

Definition 8. *(Detection Path). Given a detection rule r and its predicate graph G. $\mathcal{P}$ indicates the set of all complete paths*

in G. For a code snippet c, r is evaluated to be positive for c (i.e., $r(c) = True$) iff there exists at least one positive path $\pi \in P$ where all predicate p_v corresponding to the node v along π is positive:

$$r(c) = True \iff \exists\, \pi \in \mathcal{P} \ :\ \forall v \in \pi,\ p_v(c) = True$$

For example, the code snippet `bad2` (in Fig. 1) satisfies predicates p_{22}, p_{29}, and p_{32}, corresponding to a specific path $(\text{Start}, p_{22}, p_{29}, p_{32}, \text{End})$ within the predicate graph (Fig. 3). Consequently, the rule r would evaluate to positive for `bad2`.

Note that we normalize the rule by recursively applying De Morgan's laws [53] to eliminate negations of compound predicates. This ensures that only atomic predicates are negated, resulting in a graph free of compound-negation nodes that is more amenable to analysis. For example, $\neg((p_1 \wedge p_2) \vee p_3)$ will be normalized into $(\neg p_1 \vee \neg p_2) \wedge \neg p_3$.

2) Dynamic Profiling: Dynamic profiling captures the satisfaction status of predicates in a rule for a given code snippet t_{code}, indicating whether the code meets each predicate. Formally, predicate satisfaction facts are defined as follows:

Definition 9. *(Satisfaction Facts). Given a detection rule r (composed of predicates), a static analyzer engine $\mathcal{A}$, and a test case t with code t_{code}, the predicate satisfaction facts S_t for test case t form a mapping:*

$$S_t = \{p \to s \mid p \in r,\ s = \mathcal{A}(r, t_{code}, p)\}$$

where p is a predicate within rule r, and $s \in \{\text{true}, \text{false}\}$ is its satisfaction status for t_{code}, as determined by engine $\mathcal{A}$ when profiling predicate p.

For instance, predicate p_{25} (from Fig. 1, L_{25}) checks if `USER_DATA` (a parameter) is passed directly to the `command` function as an argument. Dynamic profiling with the Semgrep analyzer would yield $S_{\text{bad1}}(p_{25}) = \text{True}$ because the code in `bad1` exactly matches this condition (at L_5). Conversely, for `bad2`, $S_{\text{bad2}(p_{25})} = \text{False}$ as `USER_DATA` is not directly passed to `command` in that instance (at L_9).

D. Differential Fault Localization

Algorithm 1 illustrates the process of differential fault localization. We use $\mathcal{R}$ to indicate the localization results, $\mathcal{P}$ to indicate the set of paths in the predicate graph, p to indicate a predicate, and S to indicate the mapping of satisfaction facts. Function `FindAllPaths` returns all complete paths in the given graph. Function `Filter` will filter out the irrelevant paths based on the satisfaction facts (L_{15}-L_{22}). The differential comparison algorithm is implemented from L_{23}-L_{32}. The whole process can be split into two parts in general: the differential analysis (L_1-L_{12}) and the prioritization (L_{13}).

1) Differential Analysis: As mentioned in Definition 8, the detection logic can be represented as paths in the predicate graph. The key insight is that defect-revealing cases are often slight variants of regression cases, sharing most detection logic but differing in a few overlooked features due to buggy predicates. RULEREFINER leverages these differences by analyzing

Algorithm 1: Differential Fault Localization

```
Input : Defective rule r, predicate graph G, defect-revealing case ϵ
        and the regression cases T_r.
Output: Prioritized fault localizations R_prioritized.
1  R ← ∅ ;                                   // localization results
2  P_all ← FindAllPaths(G) ;                 // get all paths
3  S_ϵ ← Profiling(r, ϵ_code) ;              // satisfaction status
4  P_ϵ ← Filter(P_all, S_ϵ)
   // travesal regression cases
5  foreach t ← T_r do
6      if t_actual = ϵ_expected then
7          S_t ← Profiling(r, t_code)
8          P_t ← Filter(P_all, S_t)
           // travesal paths
9          foreach Path_ϵ ∈ P_ϵ do
10             foreach Path_t ∈ P_t do
11                 res ← Diff(Path_ϵ, Path_t, S_ϵ, S_t)
12                 R ← R ∪ {res}
13 R_prioritized ← Prioritize(R)
14 return R_prioritized

Input : A set of Paths P, a set of satisfaction status S.
Output: All positive/negative paths for positive/negative cases.
15 Function Filter(P, S)
16     P_pos ←
17       {Path ∈ P | ∀p ∈ Path, S(p) = True}
18     P_neg ← P \ P_pos
19     if P_pos ≠ ∅ then
20         return P_pos ;                    // positive case
21     else
22         return P_neg ;                    // negative case

Input : Paths under comparison Path_1, Path_2 and the
        corresponding satisfaction status S_1, S_2.
Output: Sets of intersections, intervals, and the differential results.
23 Function Diff(Path_1, Path_2, S_1, S_2)
24     intersections ← {v | v ∈ Path_1 ∩ Path_2}
25     foreach its ← intersections do
26         if S_1(its) ≠ S_2(its) then
27             diff = diff ∪ {⟨is, is⟩}
28     interval_pairs ←
         getIntervals(Path_1, Path_2, intersections)
29     foreach ⟨itv_1, itv_2⟩ ← interval_pairs do
30         if ⋀_{p∈itv_1} S_1(p) ≠ ⋀_{p∈itv_2} S_2(p) then
31             diff = diff ∪ {⟨itv_1, itv_2⟩}
32     return intersections, interval_pair, diff
```

detection paths to pinpoint the root cause, narrowing it from the entire rule to one or a few suspect predicates.

To obtain the fault localizations, RULEREFINER conducts a pair-wise comparison between every pair of paths (L_5 to L_{12}) and collects the potential buggy predicates. The implementation of the comparison on two paths is in the function `Diff` (L_{23} to L_{32}). In `Diff`, we split the path pairs into intersections and interval pairs (L_{25} and L_{28}). For each part, we compare the satisfaction facts respectively and output them if they do not match (L_{26} and L_{30}). Formally, we define the intersections and interval pairs as follows:

Definition 10. *(Intersections and Interval Pairs). For two paths P_a and P_b, its intersection $its_{\langle P_a, P_b\rangle}$ is defined as:*

$$its_{\langle P_a, P_b\rangle} := \{v \mid v \in P_a \cap P_b\}$$

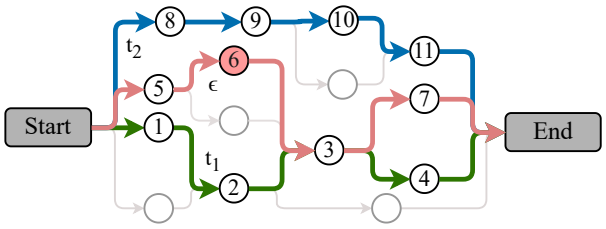


Fig. 4: Example for differential results prioritization. The green, blue, and red paths refer to the detection logic of regression case t_1, t_2, and the defect-revealing case ϵ, respectively. The numbered predicates, except the red node p_6, are all satisfied by the test case, respectively.

While the interval pairs $itv_{\langle P_a,P_b\rangle}$ is then defined as the pairs of partial paths between adjacent intersections v_i, v_j.

$$itv_{\langle P_a,P_b\rangle} \coloneqq \{\langle Interval_{P_a}(i,j), Interval_{P_b}(i,j)\rangle \mid \forall v_i, v_j \in its_{\langle P_a,P_b\rangle} \wedge adj(v_i, v_j)\}$$

*where $Interval_P(i,j)$ indicates the path slice between v_i and v_j in path P, with v_i and v_j excluded. We denote two intersections $v_i, v_j \in its_{\langle P_a,P_b\rangle}$ are **adjacent**, iff no other intersection lies between them on either path:*

$$adj(v_i, v_j) \iff v_i \prec v_j \wedge \nexists v_k \in its_{\langle P_a,P_b\rangle} : v_i \prec v_k \prec v_j$$

where $v_i \prec v_j$ indicates that v_i appers before v_j in both paths.

For instance, by comparing the paths of ϵ (red path) and t_1 (green path) in Fig. 4, we obtain three intersections, denoted as {`Start, 3, End`} and two interval pairs between them. We denote the interval pair between intersections `Start` and `3` as ⟨`(5,6), (1,2)`⟩ and the interval pairs between between intersections `3` and `End` as ⟨`(7), (4)`⟩, respectively.

We do not compare two paths by exactly matching the entire paths. Instead, `Diff` allows two cases to satisfy disjunction branches that indicate alternative detection logic to make the comparison more general in practice.

To make the comparison more efficient, we only focus on cases with the same expected detection results as the defect-revealing one (L_6). For example, we only conduct comparison on false positive cases (i.e., expected to be negative) with true negative cases. Furthermore, we also conduct filtering to exclude the irrelevant paths (L_4 and L_8) based on the satisfaction facts. Specifically, we filter out the negative paths in the positive cases (L_{20}), where the detection logic is only related to the paths where all the predicates of it are satisfied (Definition 8). While for negative cases, all paths contain unsatisfied predicates, it is non-trivial to select the actual detection logic without precise semantics information. Therefore, we keep all paths for soundness (L_{22}).

2) Prioritization: To refine the output, we select the top-N localization results after differential analysis. As mentioned above, the subtle difference can often reveal the precise root cause. Our approach prioritizes pairs with the most similar detection logic but different detection results.

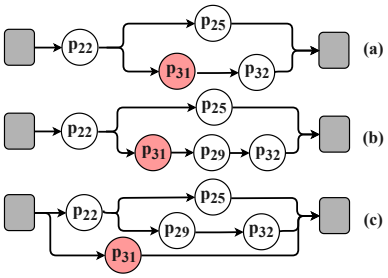


Fig. 5: Incorrect refinement.

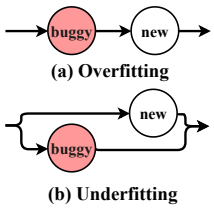


Fig. 6: Template example.

Specifically, the prioritization score for a path pair is calculated as follows:

$$score = \frac{|Intervals| + |Intersections| - |Diff|}{|Diff|}$$

where $|Intervals|$, $|Intersections|$, and $|Diff|$ denote the total number of intervals, the total number of intersections, and the number of different intervals or intersections identified by the differential analysis, respectively.

Fig. 4 showcases an example for prioritization. The green, blue, and red paths refer to the detection logic of regression case t_1, t_2, and the defect-revealing case ϵ, respectively. All the numbered predicates except the red node p_6 are satisfied by each test case. Comparing the defect-revealing case ϵ (red path) with the regression case t_1 (green path) involves two intervals and three intersections (i.e., `Start, 3, End`, with one interval with different status (i.e., ⟨`(5,6), (1,2)`⟩). This yields a prioritization score of $(2+3-1)/1 = 4$. In contrast, comparing ϵ with the regression case t_2 (blue path) involves one interval and two intersections, also with one different interval (i.e., ⟨`(8,9,10,11), (5,6,3,7)`⟩), resulting in a score of $(1+2-1)/1 = 2$. Consequently, the localization result derived from the (ϵ, t_1) pair, specifically the interval ⟨`(5,6), (1,2)`⟩ which contains the actual buggy predicate p_6, receives a higher rank and is considered first.

Algorithm 2: Template Generation Algorithm

Input : The predicate graph of defective rule g_r, satisfaction status S_ϵ and S_t, a intersection or interval in `diff`, denote as it.
Output: refinement template.

```
ph ← genPlaceHolderPredicate() // A placeholder,
     will be filled by LLM
if isIntersection(it) then
    (p, p) ← it
    if S_ϵ[p] ∧ ¬S_t[p] then
        return Replace(g_r, p, ph ∧ p)
    return Replace(g_r, p, ph ∨ p)
else
    (it_ϵ, it_t) ← it
    if ⋀_{p∈it_ϵ} S_ϵ[p] ∧ ¬ ⋀_{p∈it_t} S_e[p] then
        return Replace(g_r, it_ϵ, ph ∧ it_ϵ)
    return Replace(g_r, it_ϵ, ph ∨ it_ϵ)
```

E. Template-Based Refinement

Providing LLMs solely with the root cause information from Top-N fault localizations often proves insufficient for robust rule refinement. Due to complex and intricate dependencies between predicates, LLMs, when attempting repairs, can inadvertently undermine the carefully crafted detection logic of the original rule. Fig. 5 showcases several common incorrect refinements on predicate logic structures (the correct version is shown in Fig 3). The root causes include: (a) direct replacement of the original predicate (from p_{29} to p_{31}), leading to new false negatives; (b) faulty logical combinations (e.g., using AND instead of OR), causing broken rules; or (c) generation of overly broad, independent detection paths for the new predicate, resulting in numerous false positives.

To address these limitations, we introduce *template-based refinement*, which constrains LLMs to perform effective yet impact-limited modifications. Fig. 6 illustrates the modifications. Our approach preserves the original predicates (not shown in Fig. 6) while applying targeted, regional modifications (marked as new) around the identified root cause (marked as buggy). At a high level, the modification can be grouped into two categories: the overfitting cases and the underfitting cases. We say a predicate is:

- *underfitting* if the defect-revealing case satisfies the problematic predicate, but it is expected not to (L_4).
- *overfitting* if the defect-revealing case does not satisfy the problematic predicate, but it is expected to do so (L_6).

Based on this category, we leverage different pre-built logic skeletons to construct refinement templates, detailed in Algorithm 2. For overfitting cases, an additional predicate is inserted into the detection path after the underfitting predicate (Fig. 6a). This checkpoint is designed to filter out defect-revealing cases. For underfitting cases, a new branch is added around the buggy predicate to accommodate defect-revealing cases (i.e., false negatives), thereby preventing them from being incorrectly excluded (Fig. 6b). Specifically, given a localization result it (a potential buggy intersection or interval), the algorithm first introduces a placeholder predicate ph to denote the position for new logic (L_1, generated by function `genPlaceHolderPredicate`). Then, for underfitting predicates, a conjunctive template (e.g., $ph \wedge$ original_predicate) is generated (L_5). For overfitting predicates, a disjunctive template (e.g., $ph \vee$ original_predicate) is created (L_6). An interval containing multiple predicates is treated as a whole, applying similar generation logic (L_8-L_{11}).

Note that Fig. 2 only shows the high-level logic structure of the modification. In practice, the actual implementation includes multiple subclass templates for each category. For instance, to address overfitting rules, RULEREFINER implements six distinct subclass templates that incorporate different constraint predicates and relational operations based on the Semgrep rule grammar. Specifically, there are templates: (a) include/exclude an extra control flow pattern; (b) include/exclude an extra code context pattern; (c) include/exclude an extra data flow constraint on certain variables.

TABLE I: Rule abstraction of Semgrep

Semgrep Rule Syntax	Type	Abstration
pattern(-not)(-regex|-inside) metavariable-[pattern|regex] metavariable-[name|comparison]	Predicate	$(\neg)p$
patterns-either sources|sinks|propagators	Logic	$p \vee p$
patterns		$p \wedge p$
sanitizers		$\neg(p \vee p)$

Finally, the original buggy predicates in the predicate graph are replaced by the generated template structure. RULEREFINER generates such a template for each buggy predicate identified in the top-N prioritized results from the differential analysis.

F. LLM Refinement and Validation

Fig. 7 showcases the format of the prompt we used in RULEREFINER. After the templates with placeholders are generated, they will be translated back to the syntax of the original rule. We unite the code defective rule r, the code defect-revealing case ϵ, regression case e, the difference analysis result of them (`[Detection Difference]`), and the generated template (`[Refinement Template]`) into the prompt. We also prompt LLMs to follow a certain output format (`[Output Format]`) and think step by step (`[CoT Steps]`). Then, based on the defined output format (`[Output Format]`), RULEREFINER automatically extracts refined rule $\hat{r}$ from the response of LLMs.

The validation of the refined rule $\hat{r}$ proceeds as follows: First, $\hat{r}$ must correctly classify the defect-revealing test case ϵ, thereby confirming that the original defect has been rectified. Second, $\hat{r}$ must pass all regression test cases in the set T_r. This regression testing ensures that the refinement process has not inadvertently introduced new false positives or false negatives. RULEREFINER designates a refined rule $\hat{r}$ as successful if it satisfies both these validation criteria and outputs it accordingly. If $\hat{r}$ fails either test, RULEREFINER reports the refinement attempt as unsuccessful.

IV. EVALUATION

A. Implementation, Dataset, and Setup

1) Implementation: We select Semgrep to implement and evaluate our idea among some of the most widely used static application security testing (SAST) tools (i.e., Semgrep,

```
# Semgrep Rule Refinement Task
## Input Parameters
  [Defective Rule] + [Defect-revealing Case]
## Reference Case
  [Regression Case] + [Detection Difference]
## Task Requirements
  [Refinement Template] + [Output Format] + [CoT Steps]
```

Fig. 7: Format of refinement prompt.

TABLE II: Dataset Statistics

Dataset	Num	Pred.		Tests		Date
		Total	Avg.	Total	Avg.	
FP	131	946	7.22	1,242	9.48	2020.06.07
FN	87	870	10	970	11.15	to
Total	218	1,816	8.33	2,212	10.15	2024.10.24

CodeQL, SonarQube, PMD, CppCheck) for three main reasons: 1) ***Popularity***: Semgrep gains the most stars on GitHub among the selected SAST tools; 2) ***Significance***: more than 20K detection rules are built upon Semgrep. For example, GitLab officially supports Semgrep as the analyzer in their CI/CD pipeline. 3) ***Maintenance***: Semgrep and the community maintain an active repository of all the detection rules, from which we can obtain plenty of historical refinement commits to construct our dataset to perform evaluation. Note that the core approach of RULEREFINER (Section III) is analyzer-agnostic. We discuss the migration to other static analyzers such as PMD and Coccinelle in Section V-A.

For Semgrep rule abstraction, we employ a custom parser built on *PyYAML* to parse Semgrep rules and abstract their predicates and logical operations. The abstractions are illustrated in Table I, which contrasts the original Semgrep syntax with its abstracted forms. For fact profiling, we utilize the debugging feature provided by the Semgrep engine (`--matching-explanations`) to capture satisfaction status during detection dynamically.

2) Dataset: We collected historical rule refinements from the open-source Semgrep rule repository[1] using an automated script, identifying 371 refinement commits on the ruleset. We extracted a tuple $< r, r_{gt}, E_r, \epsilon >$ representing the original rule, the refined rule, existing tests, and a new defect-revealing test from each refinement commit and filtered out commits without regression tests or defect-revealing cases. We also manually excluded five reconstruction commits. In the end, we obtained a final dataset of 218 samples (Table II).

3) Baseline Selection and Design: Our work is the first to focus on refining complex and human-written static analysis rules via LLM. This is a new yet challenging task, and thus, there are no off-the-shelf tools to serve as baselines. Existing early works that do not leverage LLMs, such as RhoSynth [23] and SQUID [51], are designed for specific datalog-based analyzers and not open-source, adapting and comparing with them are challenging due to heavy reimplementation works.

KNighter [54] is the most relevant LLM-based approach, which employs LLMs to generate checkers of Clang Static Analyzer from code patches, and further leverages LLMs to refine the generated checkers (i.e., given the LLM-generated imperfect checker and defect-revealing case). KNighter refines checkers using *chain-of-thought* (CoT) prompting, where the LLM first generates a refinement plan before producing the final output. We adapt its refinement prompt to Semgrep as a baseline for comparison (denoted as $baseline_c$) since its target analyzer differs from ours. To systematically evaluate the impact of prompt strategies besides CoT, we further design a suite of baselines, including $baseline$ (i.e., LLM with basic prompt without complex strategies), $baseline_f$ (i.e., employing the few-shot strategy only), and $baseline_{cf}$ (i.e., employing both the few-shot and CoT strategies).

4) Setup: We set up five RQs to evaluate RULEREFINER:

- ***RQ1: Effectiveness: How many real-world defective rules can* RULEREFINER *auto-correct?***
- ***RQ2: Improvement: How does* RULEREFINER *perform comparing to the designed baselines?***
- ***RQ3: Configuration Analysis: How do the various configurations of LLMs affect the performance?***
- ***RQ4: Ablation Study: What is the contribution of each major component of* RULEREFINER*?***
- ***RQ5: Generalization: Do* RULEREFINER*-refined rules generalize similar to expert-written ones?***

We evaluate RULEREFINER and the baseline approaches using the pass@k metric [55], which measures the proportion of tasks being solved within k attempts. Specifically, for each rule, we run k independent refinements. An attempt is successful if the refined rule passes the defect-revealing test case and all regression tests (see Section III-F). The overall task is passed if any one of the k attempts is successful.

We evaluate RULEREFINER on three latest popular LLMs: *GPT-4o-mini*, *DeepSeek-V3-0324*, and *Qwen-plus-0428*. These models are all accessible via API, cost-efficient, and similarly priced (approximately $1 per million tokens). The low cost, combined with their performance, demonstrates the feasibility of our approach for practical usage at scale. Unless specifically stated, the hyperparameters are kept the same during comparisons with baselines (i.e., temperature=0.0, top_k=0.95, max_token=8192, top_p=20). In differential analysis, we select the top 5 localization results. We repeat each experiment three times and take the average as the final result. All our experiments are performed on a server with a 96-core Intel Xeon Gold 6248R CPU and 256GB of SSD memory.

B. RQ1: Effectiveness

In RQ1, we run RULEREFINER on all three selected LLMs under the default setting as mentioned above. As shown in column "RULEREFINER" in Table III, the results demonstrate the effectiveness of RULEREFINER for rule refinement. Specifically, the results demonstrate that RULEREFINER based on DeepSeek-V3 and Qwen-plus achieve a success rate of 80.28% and 73.39%, respectively. Even the lightweight GPT4o-mini model refined 108 rules, a notable success rate of 49.54%, indicating that capable performance is attainable with minimal computational resources.

The above empirical results demonstrate that RULEREFINER significantly advances the automated maintenance of static rules. RULEREFINER, empowered by cost-efficient LLMs, can successfully resolve over 80.0% of real-world defects in real-time. By automating these routine corrections, RULEREFINER enables maintainers to dedicate their expertise and efforts to the remaining minority of complex and high-value issues.

[1] https://github.com/semgrep/semgrep-rules

TABLE III: The pass@5 success rate of rule refinement on our dataset. "Basic", "Few-shot","CoT", and "CoT+Few-shot"refer to baselines with naive, Few-shot, Chain-of-Thoughts (CoT), and Few-shot plus CoT style prompts

LLM	$baseline$		$baseline_f$		$baseline_c$		$baseline_{cf}$		RULEREFINER	
GPT4o-mini	77	35.32%	66	30.28%	65	29.82%	72	33.03%	**108**	**49.54%**
Qwen-plus	86	39.45%	92	42.20%	95	43.58%	95	43.58%	**160**	**73.39%**
DeepSeek-V3	128	58.72%	141	64.68%	109	50.00%	141	64.68%	**175**	**80.28%**

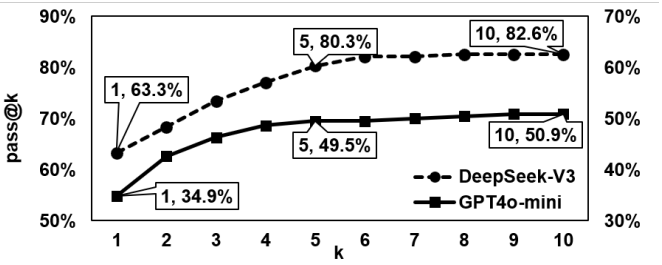


Fig. 8: The pass@k success rate results with different retry times (i.e., k).

C. RQ2: Comparision against Baselines

As shown in Table III, RULEREFINER achieves significant and consistent performance improvements over all baseline methods across each LLM. Specifically, RULEREFINER attains the highest pass@5 success rates on all three models: 49.54% on GPT4o-mini, 73.39% on Qwen-plus, and 80.28% on DeepSeek-V3. When compared to the strongest baseline for each model, RULEREFINER shows substantial gains: on GPT4o-mini, it improves upon the best baseline ($baseline$, 35.32%) by 1.40× relatively; on Qwen-plus, it outperforms the best baselines ($baseline_c$, and $baseline_{cf}$, both at 43.58%) by 1.68× relatively; and on DeepSeek-V3, it exceeds the best baselines ($baseline_f$ and $baseline_{cf}$, both at 64.68%) by 1.24× relatively. These results demonstrate that RULEREFINER consistently enhances refinement performance regardless of the underlying LLMs and prompt strategies.

In contrast, conventional prompt strategies (e.g., few-shot and CoT) yield limited improvements and even lead to performance degradation in certain scenarios. For example, on the GPT4o-mini model, both few-shot and CoT prompting (i.e., $baseline_f$, $baseline_c$, and $baseline_{cf}$) perform worse than naive prompting (i.e., $baseline$). A performance drop is also observed with the CoT baseline (i.e., $baseline_c$) on DeepSeek-V3. We attribute these results to two main factors. First, the limited size of GPT4o-mini makes it prone to overfitting to the specific refinement strategies presented in the few-shot examples, thereby impairing its generalization capability. Second, for CoT baselines (i.e., $baseline_c$ and $baseline_{cf}$), the lack of fact-based grounding allows hallucinations to emerge and accumulate during extended reasoning chains, ultimately compromising refinement performance [56].

D. RQ3: Configuration Analysis

In this RQ, we evaluate the impact of temperature settings and retry times of RULEREFINER, which are the two key settings that can be configured in RULEREFINER. Fig. 8 showcases the pass@k results under different retry times (i.e., K). The results show significant performance gains in the first several rounds of retries. However, the returns diminish, with subsequent retries providing only marginal improvement. It shows that the results in RQ1 can be further improved with more retries (e.g., 82.6% with 10 retries on DeepSeek). However, the computational cost nearly doubles. Therefore, for practical use, we recommend setting a retry limit of 4-6 to optimally balance performance and cost for the refinement tasks.

TABLE IV: The pass@5 results with different temperatures

LLM	Temperature					
	0.0		0.5		1.0	
GPT4o-mini	108	49.54%	120	55.05%	**127**	**58.26%**
Qwen-plus	**160**	**73.39%**	158	72.48%	158	72.48%
Deepseek-V3	**175**	**80.28%**	170	77.98%	169	77.52%

As shown in Table IV, the effect of temperature varies across the different models. For the larger models (e.g., Qwen and DeepSeek), a temperature of zero yields slightly better pass@5 scores. In contrast, the smaller GPT-4o-mini model performs best at a higher temperature of 1.0. We interpret that this discrepancy stems from model capability. For the DeepSeek and Qwen models, the most confident responses generated under zero temperature are more likely to be correct; therefore, a higher temperature increases the risk of bypassing the correct solution within the five attempts. However, GPT-4o-mini, which has relatively weaker reasoning and understanding abilities, benefits from the increased diversity, sampling from a broader range of potential outputs, thereby improving the chance of generating a correct solution.

E. RQ4: Ablation Study

Table V shows the results of our ablation study, which clearly demonstrate the individual and combined contributions of *differential fault localization* and *template-based refinement* to the overall performance of RULEREFINER.

Specifically, across all three LLMs, there is a consistent and significant improvement in the pass@1 rate as more components of RULEREFINER are enabled. For instance, with Deepseek, RULEREFINER-LT (lacking both differential localization and template-based refinement) achieves a 32.57% pass rate. Adding differential localization (RULEREFINER-T) boosts this to 56.88%, and further incorporating template-based refinement elevates the performance to 62.84%. This trend underscores the critical roles of both differential fault

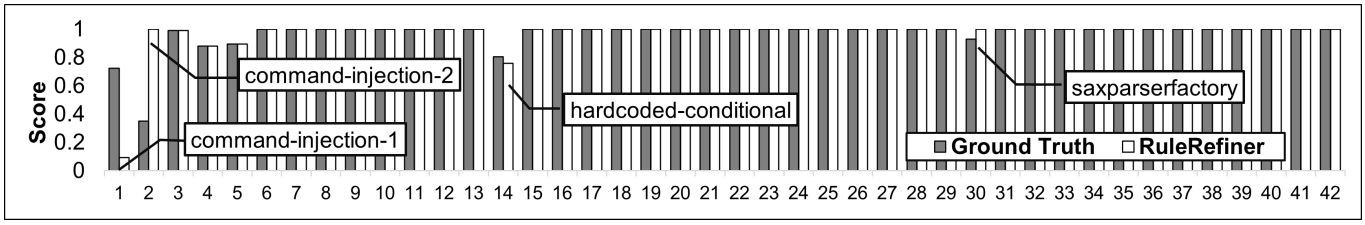


Fig. 9: Comparison of generalization scores between expert refined (i.e., Ground Truth) and RULEREFINER-refined Java rules on defect-revealing variants. The X-axis represents different defect rules, and the Y-axis represents the generalization score.

TABLE V: The pass@1 result of the ablation study, "RULEREFINER-LT", "RULEREFINER-T", "RULEREFINER" indicate RULEREFINER without differential localization and template-based refinement, without template-based refinement only and RULEREFINER with all features, respectively.

LLM	RULEREFINER-LT		RULEREFINER-T		RULEREFINER	
GPT4o-mini	53	24.31%	57	26.15%	**76**	**34.86%**
QWEN	55	25.23%	96	44.04%	**115**	**52.75%**
Deepseek	71	32.57%	124	56.88%	**137**	**62.84%**

localization in accurately identifying the root cause of the buggy rule predicate and template-based refinement in guiding the LLM to make precise and effective corrections.

F. RQ5: Generalization

In RQ5, we evaluate the generalization capability of rules refined by RULEREFINER in comparison to expert-refined ground truth rules. Specifically, we employ Statfier [50], a state-of-the-art testing tool for static analyzer rules, to generate semantic-preserving variants and assess the pass rate of each refined rule over these variants. A higher pass rate indicates stronger generalization, as the rule can correctly identify more semantically equivalent buggy code patterns. We then compare the scores between RULEREFINER-refined rules and expert-written rules. If the scores are comparable, we conclude that the corresponding rules exhibit similar generalization capability in detecting buggy code variants. Since Statfier is designed for Java, we restrict this experiment to Java rules. As presented in Table VI, the evaluation covers 126 test cases and 207K generated variants in total. The results demonstrate that the generalization performance of RULEREFINER-refined rules is highly aligned with that of expert-written rules. Most notably, RULEREFINER achieves identical performance to the ground truth on regression test variants (98.08%), while even slightly surpassing expert performance on defect-revealing variants (96.68% vs. 96.61%). This indicates that RULEREFINER can indeed refine rules while not breaking the regression functional equivalence and generating well-generalized patterns as the expert-written ones.

To further interpret the above results, we conduct a fine-grained analysis of the results on the defect-revealing variants. Fig. 9 illustrates the generalization score for each defective rule across all 42 defect-revealing test variants. The results indicate that for the vast majority of rules, both the expert-written and RULEREFINER-refined rules correctly handle all generated variants. Only a small number of rules (4 out of 42) exhibit inconsistent behaviors, as marked in Fig. 9. Among these, only one case demonstrates a substantial decrease in generalization capability compared to the expert-refined rule (i.e., `command-injection-1`). Our manual inspection reveals that for this case, the LLM overfits a specific syntactic pattern in the defect-revealing example, thus resulting in a reduced score. However, the overfitting is rare (2.4%) and can be readily addressed by incorporating failing test cases into a subsequent refinement iteration.

TABLE VI: The testing results of RQ5. "Num." indicates the number of test cases, "Variant Num." indicates the number of variants generated for generalization evaluation.

Category	Num.	Variant Num.	Ground Truth	RULEREFINER
Defect-revealing	42	51K	96.61%	96.68%
Regression	84	156K	98.08%	98.08%
Total	126	207K	97.71%	97.73%

```
1  * if (E == NULL)
2  { ... when != if (E == NULL) S1 else S2
3        when != E = E1
4  (
5  * E->f
6  |
7  * DEREF(E)
8  )
9  ... when any
10 } else S3
```

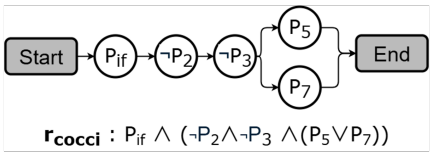


Fig. 10: Rule abstraction of a Coccinelle rule, the metavariable definitions are omitted for simplification.

V. DISCUSSION

A. Adaptation To Other Analyzers

To adapt RULEREFINER for static analyzers beyond Semgrep, two primary components require careful modification: rule abstraction and the dynamic profiling interface.

Rule abstraction involves translating diverse rule languages into the standardized format RULEREFINER utilizes, as detailed in Section III-A1. It is important to note that this process does not require capturing all grammatical details of the rule or the underlying matching mechanisms of the analysis engine. Instead, for adaptation purposes, the emphasis lies on selecting a suitable level of abstraction that enables the decomposition of the entire detection rule into verifiable

elements (i.e., predicates) and supports the modeling of logical relationships among them. Following this principle, we treat these elements as atomic units, abstracting away any internal logic or complex grammar they might contain.

For example, Coccinelle [57] is a program matching and transformation tool widely used for evolving the Linux kernel. As illustrated in Fig. 10, rules in Coccinelle are written in the SmPL[2] language and the rule is used to detect null pointer dereferences in C programs. Specifically, SmPL uses "`( p_1 |...| p_n )`" to indicate the disjunction of patterns (e.g., L_4-L_8), and uses "`when !=`" to specify the whitelist pattern (e.g., L_2). Besides, sequential statements are treated as conjunctions. Thus, the rule in Fig. 10 can be translated to $p_{if} \wedge (\neg p_2 \wedge \neg p_3 \wedge (p_5 \vee p_6))$, where p_{if} indicates the pattern cross L_1 to L_{10}, which matches the outermost if-else branch. The graph representation is also illustrated in Fig. 10.

Moreover, for rules written in *general-purpose languages* (GPLs) (e.g., CSA or PMD), predicates are typically identified within the branch conditions and assertions. The predicates and logical flow can be extracted by analyzing the control flow graph of the code that defines the rule.

The dynamic profiling interface, crucial for RULEREFINER to understand rule behavior, also needs analyzer-specific tailoring. For DSL-based rules, profiling might necessitate support from the analyzer engine to expose predicate satisfaction facts. For GPL-based rules, profiling information can often be obtained by instrumenting the code to capture execution traces and correlating this data (like condition coverage) back to specific rule elements.

B. Failures in Refinement

Although our evaluation shows that RULEREFINER can successfully handle 70%–80% of real-world refinement tasks using DeepSeek and Qwen within five attempts, there are also challenging cases that RULEREFINER cannot yet automatically correct due to the complexity of real-world issues.

First, rules with inadequate test suites hinder fault localization, reducing performance. Second, our template-based approach cannot manage rules needing complete redesign, such as shifting from pattern-matching to taint analysis. Third, LLMs sometimes misunderstand vulnerabilities or struggle with complex grammar, leading to overly specific or syntactically incorrect rules. We plan to mitigate these by using LLMs to generate tests and integrating RAG to provide richer context. To address this, we plan to integrate more context (grammar documents, CVE details, and additional examples) and use Retrieval-Augmented Generation (RAG) techniques to improve the performance in our future work.

VI. Related Work

A. Static Analyzer False Alarm Mitigation

Research in false alarm mitigation employs two main strategies. The first enhances the static analyzer itself, using advanced algorithms or, more recently, LLMs to infer specifications and prune paths for improved precision [28], [29], [39], [58]–[60]. The second strategy filters warnings after they are generated, using historical data to classify them or leveraging LLMs to identify false positives directly from warning lists [27], [40], [61]. A key limitation of these post-hoc methods is that they do not fix the analyzer's rules and can introduce false negatives. In contrast, RULEREFINER addresses the root cause by directly refining the defective ruleset to eliminate both false positives and false negatives.

B. Static Analysis Detection Rule Generation

A distinct area of research explores the generation or synthesis of new static analyzer detection rules, often from code examples or specifications [62]. For example, RhoSynth [23] performs rule synthesis on graph representations of code using a novel graph alignment algorithm. SQUID [51] synthesizes code search queries for Datalog-based analyzers from positive and negative examples, employing pruning techniques to manage the search space. KNighter [54] utilizes LLMs to generate checkers for the Clang Static Analyzer from code patches, although Knighter also leverages LLMs to refine the generated rules, but the refinement is conducted directly by querying LLMs (as our basic prompt baseline). These works primarily concentrate on creating new static analyzer rules from scratch or based on provided exemplars. This differs fundamentally from our work with RULEREFINER, which focuses on the *refinement* of *existing*, potentially complex and human-written, rules to correct identified false positives and/or false negatives.

VII. Conclusion

This paper proposes RULEREFINER, a pioneering framework for automatic refinement of defective static analysis rules. RULEREFINER empowers LLMs to directly refine static analysis rules by dynamic profiling, differential fault localization, and template-based refinement. We evaluate RULEREFINER on the real-world rule refinement issues, the evaluation results demonstrate promising performance of RULEREFINER, achieving up to a success rate of 80.28% and substantially outperforming baseline LLM prompting strategies. Furthermore, rules refined by RULEREFINER exhibited generalization capabilities comparable to those crafted by human experts.

Acknowledgement

We sincerely thank all anonymous reviewers for their valuable feedback and guidance in improving this paper. This work was sponsored by the Major Program (JD) of Hubei Province (No.2023BAA024).

References

[1] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," in *Empir. Softw. Eng.*, vol. 25, no. 2, 2020, pp. 1419–1457. [Online]. Available: https://doi.org/10.1007/s10664-019-09750-5

[2] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, pp. 62–70, 2019. [Online]. Available: https://doi.org/10.1145/3338112

[2] https://coccinelle.gitlabpages.inria.fr/website/docs/main_grammar.html

[3] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018. [Online]. Available: https://doi.org/10.1145/3188720

[4] J. Bai, "BESA: extending bugs triggered by runtime testing via static analysis," in *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, March 30-April 3, 2025*. ACM, 2025, pp. 1077–1091. [Online]. Available: https://doi.org/10.1145/3689031.3696089

[5] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*. ACM, 2021, pp. 2183–2196. [Online]. Available: https://doi.org/10.1145/3460120.3484541

[6] M. Cui, C. Chen, H. Xu, and Y. Zhou, "Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 82:1–82:21, 2023. [Online]. Available: https://doi.org/10.1145/3542948

[7] J. Yan, J. Huang, C. Fang, J. Yan, and J. Zhang, "Better debugging: Combining static analysis and llms for explainable crashing fault localization," *CoRR*, vol. abs/2408.12070, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.12070

[8] A. Mordahl, Z. Zhang, D. Soles, and S. Wei, "ECSTATIC: an extensible framework for testing and debugging configurable static analysis," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 550–562. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00056

[9] A. Mordahl, D. Soles, M. Miao, Z. Zhang, and S. Wei, "ECSTATIC: automatic configuration-aware testing and debugging of static analysis tools," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. ACM, 2023, pp. 1479–1482. [Online]. Available: https://doi.org/10.1145/3597926.3604918

[10] J. Bai, J. Lawall, Q. Chen, and S. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019, pp. 255–268. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/bai

[11] Sonarqube. [Online]. Available: https://www.sonarsource.com/

[12] Errorprone. [Online]. Available: https://github.com/google/error-prone

[13] Infer. [Online]. Available: https://github.com/facebook/infer

[14] Semgrep. [Online]. Available: https://github.com/semgrep/semgrep

[15] Codeql. [Online]. Available: https://github.com/github/codeql

[16] Pmd. [Online]. Available: https://github.com/github/pmd

[17] B. Aloraini, M. Nagappan, D. M. Germán, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *J. Syst. Softw.*, vol. 158, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.110427

[18] J. Yang, L. Tan, J. Peyton, and K. A. Duer, "Towards better utilizing static application security testing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 51–60. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00014

[19] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, ""false negative - that one is going to kill you": Understanding industry perspectives of static analysis based security testing," in *Proceedings of IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 3979–3997. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00019

[20] W. He, P. Di, M. Ming, C. Zhang, T. Su, S. Li, and Y. Sui, "Finding and understanding defects in static analyzers by constructing automated oracles," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1656–1678, 2024. [Online]. Available: https://doi.org/10.1145/3660781

[21] M. Fourné, D. D. A. Braga, J. Jancar, M. Sabt, P. Schwabe, G. Barthe, P. Fouque, and Y. Acar, ""these results must be false": A usability evaluation of constant-time analysis tools," in *Proceedings of the 33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/fourne

[22] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 698–709. [Online]. Available: https://doi.org/10.1145/3510003.3510214

[23] P. Garg and S. H. Sengamedu, "Synthesizing code quality rules from examples," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 1757–1787, 2022. [Online]. Available: https://doi.org/10.1145/3563350

[24] G. D. Palma, S. Giallorenzo, C. Laneve, J. Mauro, M. Trentin, and G. Zavattaro, "Leveraging static analysis for cost-aware serverless scheduling policies," *Int. J. Softw. Tools Technol. Transf.*, vol. 26, no. 6, pp. 781–796, 2024. [Online]. Available: https://doi.org/10.1007/s10009-024-00776-9

[25] W. Zheng, Y. Jiang, and X. Su, "Vu1spg: Vulnerability detection based on slice property graph representation learning," in *Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*. IEEE, 2021, pp. 457–467. [Online]. Available: https://doi.org/10.1109/ISSRE52982.2021.00054

[26] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Trans. Software Eng.*, vol. 49, no. 12, pp. 5154–5188, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3329667

[27] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2017, Barcelona, Spain, June 18, 2017*. ACM, 2017, pp. 35–42. [Online]. Available: https://doi.org/10.1145/3088525.3088675

[28] J. Giet, L. Mauborgne, D. Kästner, and C. Ferdinand, "Towards zero alarms in sound static analysis of finite state machines," in *Proceedings of the 38th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2019, Turku, Finland, September 11-13, 2019*, ser. Lecture Notes in Computer Science, vol. 11698. Springer, 2019, pp. 3–18. [Online]. Available: https://doi.org/10.1007/978-3-030-26601-1_1

[29] T. Muske, R. Talluri, and A. Serebrenik, "Reducing static analysis alarms based on non-impacting control dependencies," in *Proceedings of the 17th Asian Symposium on Programming Languages and Systems, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019*, ser. Lecture Notes in Computer Science, vol. 11893. Springer, 2019, pp. 115–135. [Online]. Available: https://doi.org/10.1007/978-3-030-34175-6_7

[30] Y. Yang, M. Wen, X. Gao, Y. Zhang, and H. Sun, "Reducing false positives of static bug detectors through code representation learning," in *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*. IEEE, 2024, pp. 681–692. [Online]. Available: https://doi.org/10.1109/SANER60148.2024.00075

[31] H. Kim, M. Raghothaman, and K. Heo, "Learning probabilistic models for static analysis alarms," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1282–1293. [Online]. Available: https://doi.org/10.1145/3510003.3510098

[32] A. Murali, N. S. Mathews, M. Alfadel, M. Nagappan, and M. Xu, "Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 65:1–65:13. [Online]. Available: https://doi.org/10.1145/3597503.3623321

[33] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic root cause analysis via large language models for cloud incidents," in *Proceedings of the 19th European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 2024, pp. 674–688. [Online]. Available: https://doi.org/10.1145/3627703.3629553

[34] Z. Jiang, M. Wen, J. Cao, X. Shi, and H. Jin, "Towards understanding the effectiveness of large language models on directed test input generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 2024, pp. 1408–1420. [Online]. Available: https://doi.org/10.1145/3691620.3695513

[35] Y. Wu, M. Wen, Z. Yu, X. Guo, and H. Jin, "Effective vulnerable function identification based on CVE description empowered by large

language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 2024, pp. 393–405. [Online]. Available: https://doi.org/10.1145/3691620.3695013

[36] Z. Yu, Y. Zhang, M. Wen, Y. Nie, W. Zhang, and M. Yang, "Cxxcrafter: An llm-based agent for automated C/C++ open source software building," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 2618–2640, 2025. [Online]. Available: https://doi.org/10.1145/3729386

[37] Z. Yu, M. Wen, X. Guo, and H. Jin, "Maltracker: A fine-grained NPM malware tracker copiloted by llm-enhanced dataset," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*. ACM, 2024, pp. 1759–1771. [Online]. Available: https://doi.org/10.1145/3650212.3680397

[38] X. Du, M. Wen, J. Zhu, Z. Xie, B. Ji, H. Liu, X. Shi, and H. Jin, "Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning," in *Proceedings of Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*. Association for Computational Linguistics, 2024, pp. 10 507–10 521. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.625

[39] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 2023, pp. 2107–2111. [Online]. Available: https://doi.org/10.1145/3611643.3613078

[40] J. Chen, H. Xiang, L. Li, Y. Zhang, B. Ding, and Q. Li, "Utilizing precise and complete code context to guide LLM in automatic false positive mitigation," *CoRR*, vol. abs/2411.03079, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.03079

[41] H. Cui, M. Xie, T. Su, C. Zhang, and S. H. Tan, "An empirical study of false negatives and positives of static code analyzers from the perspective of historical issues," 2024. [Online]. Available: https://arxiv.org/abs/2408.13855

[42] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. R. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 4:1–4:13. [Online]. Available: https://doi.org/10.1145/3597503.3608132

[43] J. Xiao, Z. Xu, S. Chen, G. Lei, G. Fan, Y. Cao, S. Deng, and Z. Feng, "Confix: Combining node-level fix templates and masked language model for automatic program repair," *J. Syst. Softw.*, vol. 216, p. 112116, 2024. [Online]. Available: https://doi.org/10.1016/j.jss.2024.112116

[44] X. Meng, X. Wang, H. Zhang, H. Sun, X. Liu, and C. Hu, "Template-based neural program repair," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1456–1468. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00127

[45] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 535–547. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00063

[46] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 2019, pp. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[47] Y. Zhang and Y. Wang, "Setemapr: Incorporating semantic knowledge in template-based neural program repair," in *Proceedings of the International Joint Conference on Neural Networks, IJCNN 2024, Yokohama, Japan, June 30 - July 5, 2024*. IEEE, 2024, pp. 1–8. [Online]. Available: https://doi.org/10.1109/IJCNN60899.2024.10650123

[48] T. Nguyen, Q. Ta, and W. Chin, "Automatic program repair using formal verification and expression templates," in *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2019, Cascais, Portugal, January 13-15, 2019*, ser. Lecture Notes in Computer Science, vol. 11388. Springer, 2019, pp. 70–91. [Online]. Available: https://doi.org/10.1007/978-3-030-11245-5_4

[49] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," *CoRR*, vol. abs/1903.08409, 2019. [Online]. Available: http://arxiv.org/abs/1903.08409

[50] H. Zhang, Y. Pei, J. Chen, and S. H. Tan, "Statfier: Automated testing of static analyzers via semantic-preserving program transformations," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 2023, pp. 237–249. [Online]. Available: https://doi.org/10.1145/3611643.3616272

[51] C. Wang, P. Yao, W. Tang, G. Fan, and C. Zhang, "Synthesizing conjunctive queries for code search," in *Proceedings of the 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, ser. LIPIcs, vol. 263, 2023, pp. 36:1–36:30. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2023.36

[52] L. N. Q. Do and E. Bodden, "Explaining static analysis with rule graphs," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 678–690, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.2999534

[53] A. De Morgan, *Formal logic: or, the calculus of inference, necessary and probable*. Taylor and Walton, 1847.

[54] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang, "Knighter: Transforming static analysis with llm-synthesized checkers," *CoRR*, vol. abs/2503.09002, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.09002

[55] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[56] X. Li, Z. Yu, Z. Zhang, X. Chen, Z. Zhang, Y. Zhuang, N. Sadagopan, and A. Beniwal, "When thinking fails: The pitfalls of reasoning for instruction-following in llms," *CoRR*, vol. abs/2505.11423, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.11423

[57] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the linux kernel," in *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 601–614. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/lawall

[58] G. Liang, Q. Wu, Q. Wang, and H. Mei, "An effective defect detection and warning prioritization approach for resource leaks," in *Proceedings of the 36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*. IEEE Computer Society, 2012, pp. 119–128. [Online]. Available: https://doi.org/10.1109/COMPSAC.2012.22

[59] Z. Li, S. Dutta, and M. Naik, "IRIS: llm-assisted static analysis for detecting security vulnerabilities," in *Proceedings on the 13th International Conference on Learning Representations,ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=9LdJDU7E91

[60] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, pp. 474–499, 2024. [Online]. Available: https://doi.org/10.1145/3649828

[61] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: improving actionable alert ranking," in *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM, 2014, pp. 152–161. [Online]. Available: https://doi.org/10.1145/2597073.2597100

[62] C. Latappy, Q. Perez, T. Degueule, J. Falleri, C. Urtado, S. Vauttier, X. Blanc, and C. Teyton, "Mlinter: Learning coding practices from examples - dream or reality?" in *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*. IEEE, 2023, pp. 795–804. [Online]. Available: https://doi.org/10.1109/SANER56733.2023.00092