


Divide and Conquer: Exposing SAST Rule Flaws via Realistic Variant Synthesis from Millions of Repositories

Zongze Jiang*†
Huazhong University of Science and Technology
Wuhan, China
jiangzongze@hust.edu.cn

Ming Wen*†‡
Huazhong University of Science and Technology
Wuhan, China
mwenaa@hust.edu.cn

Zifan Xie
Chongqing University
Chongqing, China
xzff@cqu.edu.cn

Yuan Zhang
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Deqing Zou*†
Huazhong University of Science and Technology
Wuhan, China
deqingzou@hust.edu.cn

Hai Jin*§
Huazhong University of Science and Technology
Wuhan, China
hjin@hust.edu.cn

Abstract

The effectiveness of *Static Application Security Testing* (SAST) tools hinges on the precision of their detection rules. Yet, refining these rules is critically bottlenecked by the scarcity of realistic test cases. Existing approaches present a fundamental trade-off: mining-based methods yield realistic but sparse samples of high-severity vulnerabilities, while mutation-based generators achieve scale but produce syntactically legal yet unnatural constructs that are of limited use in practical application scenarios.

We present S-MRVA, a novel paradigm that generates realistic security test cases via a divide-and-conquer strategy. By leveraging semantic decomposition, targeted retrieval and synthesis, and an LLM-as-a-Judge differential oracle, S-MRVA synthesizes real-world grounded variants from GitHub repositories to effectively expose flaws in SAST rules. Evaluated across 220 rules from four mainstream SAST tools, S-MRVA uncovers over 400 false positives and false negatives while achieving 97.27% rule coverage, substantially outperforming the baselines. To demonstrate real-world impact, we have submitted 89 unique rule defects across 43 reports to the respective maintainers. Maintainers have reported that rules refined using our generated test cases have successfully identified new vulnerabilities in the wild.

CCS Concepts

• **Security and privacy → Software security engineering**.

*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
†Hubei Engineering Research Center on Big Data Security, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, HUST, Wuhan, 430074, China
‡Corresponding author.
§Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China



This work is licensed under a Creative Commons Attribution 4.0 International License.

CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846784

Keywords

SAST, Software Testing, Variant Synthesis, Large Language Models

ACM Reference Format:
Zongze Jiang, Ming Wen, Zifan Xie, Yuan Zhang, Deqing Zou, and Hai Jin. 2026. Divide and Conquer: Exposing SAST Rule Flaws via Realistic Variant Synthesis from Millions of Repositories. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846784

1 Introduction

Static Application Security Testing (SAST) tools form a foundational layer of security assurance in modern software development lifecycles and DevSecOps pipelines [21, 22, 45]. These tools include commercial platforms such as Checkmarx [4], Fortify [28], and SonarQube [36], as well as open-source or free tools such as CodeQL [10], Semgrep [34], and Bandit [33]. Deployed across IDEs, CI/CD workflows, and code review platforms, they continuously inspect source code and have detected numerous critical security flaws before software reaches production [22, 35]. For software maintainers, a failure of these tools means vulnerabilities do not simply remain as latent defects, but may become exploitable attack surfaces that propagate directly into deployed systems.

Naturally, this raises a pressing question: *what causes SAST tools to fail?* The accuracy of modern SAST tools depends on two essential components: the *backend analysis engine* and the *detection rules* [17]. The engine provides core program analysis capabilities (e.g., taint tracking and symbolic execution), while the rules encode concrete vulnerability patterns that the engine is equipped to detect. Although both components contribute to SAST accuracy, engine-level advances are often constrained by fundamental program-analysis challenges, such as alias resolution, and practical performance requirements in CI/CD settings. Consequently, refining detection rules offers a more practical and immediate solution towards improving security assurance.

We use the Log4j vulnerability[1] to demonstrate why such refinement is urgently needed. The CodeQL engine could have detected this critical flaw before public disclosure, yet its detection rules did

[1] https://www.cve.org/CVERecord?id=CVE-2021-44228

not fully model the relevant exploitable pattern at the time. As a result, the tool produced false negatives, allowing the vulnerability to escape detection and ultimately enabling widespread exploitation. The CodeQL team's retrospective analysis acknowledged this gap.[2]

> *"Therefore, CodeQL could have reported Log4Shell if we had analyzed an application that took untrusted data from the network (for example, a web application) and passed this untrusted data to Log4J logger methods."*

SAST rule development is expected to follow test-driven development principles, where each detection pattern requires corresponding test cases to guard against regressions and guide new pattern creation. In practice, however, such test cases are scarce. Compared to the thousands of rules these tools maintain (e.g., over 2,000 for CodeQL), these tools' validation suites remain severely insufficient, relying largely on handcrafted test cases derived from manual vulnerability reverse-engineering, CVE scraping, and expert domain knowledge. This labor-intensive process cannot feasibly cover the combinatorial explosion of framework-specific APIs, language variants, and novel vulnerability classes, leaving false positives and negatives pervasive in practice [1, 13, 43].

To overcome these limitations, recent work has leveraged automated techniques to collect or generate vulnerability variants as test cases. Industrial approaches like MRVA [12] mine vulnerability variants directly across real-world repositories. However, MRVA only discovers variants that the rules already cover. Also, the long-tail distribution of vulnerabilities (mostly low severity) and labor-intensive manual auditing limit its effectiveness in surfacing refinable cases. Other works automate test generation [8, 14, 20, 25, 42], including random generation [8], metamorphic transformations [42], and fault injection [14], yet they assume SAST rules must handle ***any*** syntactically valid code. Such risks misaligning with the actual purpose of SAST: detecting security anti-patterns that developers ***realistically*** introduce. Specifically, these approaches generate edge-case variants (e.g., wrapping vulnerabilities in `always-false` branches) that, while technically valid, often do not reflect real-world vulnerability patterns. This concern is also reflected repeatedly in open-source maintainer feedback on such variants. CodeQL maintainers have commented as follows.[3]

> *"We will probably not prioritize this until we start seeing instances of this in the wild."*

Similarly, Checkstyle maintainers have remarked as follows.[4]

> *"In real life such code is highly unlikely to exist."*

These responses strongly suggest that automated testing detached from realism may offer limited help in rule development. In fact, it actively increases the maintenance burden.

In summary, *existing approaches face a fundamental trade-off*: mining-based methods like MRVA yield highly realistic samples but struggle to generate sufficient test cases for high-severity vulnerabilities, whereas heuristic generation tools [8, 14, 42] like Statfier achieve large scale but lack grounding in real-world scenarios. To resolve this, we analyzed hundreds of SAST rules in CodeQL and Semgrep. Specifically, for each rule, we collected both the rule implementations and the vulnerable code instances they are designed to capture. From this analysis, we observe two key characteristics: First, *SAST rules fundamentally comprise multiple semantically independent sub-patterns, mirroring the inherent modularity of the real-world vulnerabilities they target.* While most rules adhere to this modular paradigm, taint-analysis rules represent the most extreme manifestation. They are explicitly decomposed into obligatory components (i.e., sources, sinks, sanitizers, and propagators), each directly mapped to a distinct functional element of taint-style vulnerabilities. Second, *although fully formed vulnerability instances are rare in the wild, the code snippets corresponding to these constituent sub-patterns are abundant and widely distributed across real-world codebases.* For instance, while our mining of the top 1K GitHub repositories with CodeQL identified only a few SQL Injection vulnerabilities, we found thousands of diverse and realistic instances of code that handle *user input* or *database queries*.

These observations motivate a ***divide-and-conquer*** strategy that balances realism and scalability. Rather than mining for complete vulnerability instances (rare) or mutating existing tests syntactically (sacrificing realism), we synthesize variants from real-world instances. Specifically, we first perform semantic decomposition (see Section 4.1) to refine SAST rules into sub-patterns with balanced granularity, enabling each sub-pattern to be matched against abundant real-world code. Second, we conduct targeted retrieval and variant synthesis (see Section 4.2) to retrieve diverse yet semantically relevant code from real-world repositories for each decomposed sub-pattern independently, and integrate the retrieved code into the existing seed to construct ***real-world grounded*** variants. Finally, we apply validation and root-cause clustering (see Section 4.3) using a rule-specific LLM-as-a-Judge module as a differential testing oracle to detect potential flaws in the rules.

We implement our idea as an agentic workflow named S-MRVA. Specifically, built upon GitHub code search,[5] S-MRVA can automatically generate and validate variants synthesized from over 200M public repositories. Evaluated on 220 rules spanning four popular SAST tools, S-MRVA detected 458 false positives and false negatives, covering 97.27% of rules and outperforming the mutation-based and mining-based baselines. Moreover, the generated real-world grounded variants exhibit significantly higher naturalness and diversity, confirming that the approach simultaneously achieves realism and scale. We have filed 89 high-severity findings across 43 reports, and 20 of the 21 responded-to reports have been confirmed by maintainers. Industrial security teams further report that our variants enabled the interception of new true vulnerabilities and the reduction of false alerts in real-world scanning.

In summary, this paper makes the following contributions:

- **Problem Characterization**: We formalize the tension between *realism* and *scalability* in SAST testing, and empirically establish that while complete vulnerabilities are rare, their semantic sub-patterns are abundant in the wild.
- **Divide-and-Conquer Paradigm**: We propose a novel testing approach that decomposes SAST rules into semantic sub-patterns, mines realistic code snippets, and synthesizes them to generate ***real-world grounded variants*** at scale.

[2] https://github.blog/security/vulnerability-research/announcing-codeql-community-packs/
[3] https://github.com/github/codeql/issues/21526
[4] https://github.com/checkstyle/checkstyle/issues/11580
[5] https://github.com/features/code-search

- **System Implementation**: We implement this paradigm in **S-MRVA**, an agentic workflow that validates SAST rules against effective variants synthesized with code from over 200 million repositories automatically.
- **Empirical Evaluation**: We evaluate S-MRVA on 220 rules across four mainstream SAST tools and three languages, uncovering 458 false positives/negatives with 97.27% rule coverage, significantly outperforming the baselines.
- **Community Impact**: We report 89 high-severity cases to maintainers with over a 90% confirmation rate. Industrial security teams report that our variants enabled the interception of new true vulnerabilities and reduced false alerts in production scanning.

The artifact is available at: https://github.com/CGCL-codes/S-MRVA for future research.

2 Motivation and Challenges

Figure 1(a) shows a simplified CodeQL rule for Time-of-Check to Time-of-Use (TOCTOU) vulnerabilities, which can lead to privilege escalation and file corruption on Unix-like systems [27]. The CodeQL rule models three components: a temporary file creation, a file operation, and a data-flow path connecting them without safe interception. Figure 1(b) shows a vulnerable instance it captures. However, the rule is imperfect. Figure 1(f) shows a real-world false negative. The false negative code uses `NamedTemporaryFile` with a `delete=False` configuration, stores the filename, and later operates on it, resulting in a TOCTOU vulnerability.

Existing approaches cannot generate such a test case to expose the rule defect. Mining-based methods like MRVA are *reactive*. They only harvest instances that current rules already match, missing false negatives entirely. Mutation-based methods like Statfier apply syntactic transformations (e.g., wrapping code in `always-false` branches as shown in Figure 1(d)) but struggle to synthesize functionally meaningful alternatives that represent real anti-patterns.

Our key insight is twofold. First, SAST rules decompose naturally into independent *semantic sub-patterns* (highlighted by color in Figure 1(a)). Second, while complete vulnerability instances are scarce, code snippets matching *individual* sub-patterns are abundant. For instance, mining the top 1K GitHub repositories for TOCTOU yielded only a few already-matched vulnerabilities, but thousands of realistic instances of temporary file creation. S-MRVA operationalizes this insight via a ***divide-and-conquer*** strategy. Specifically, it decomposes the rule into sub-patterns (Figure 1(a)), mines real-world code for each sub-pattern in (c), and compositionally synthesizes novel variants from these building blocks shown in (e). The resulting variant directly exposes the gap, enabling the refined rule to capture the real-world false negative in (f).

While the core intuition is straightforward, realizing this divide-and-conquer strategy introduces three technical challenges that span SAST rule analysis, large-scale code retrieval, and oracle design. We describe each challenge and our solution below.

Challenge 1: Decomposing Rules into Appropriate Granularity. As shown in Figure 1, the granularity of rule decomposition fundamentally determines the quality of subsequent retrieval and synthesis. However, the wide structural diversity among SAST analyzers makes simple decomposition heuristics (e.g., syntactic methods) prone to failure, as they tend to produce either overly fragmented or overly coarse outputs. Over-fragmentation breaks the rule into isolated sub-patterns that lose semantic local context, making it difficult to retrieve code matching the rule's actual intent. Conversely, an overly coarse decomposition embeds too much context into a single sub-pattern, causing the generated query to overfit the seed and reducing the diversity of retrieved candidates.

Solution. Our key idea is to perform decomposition based on semantic behavior rather than syntactic structure (Section 4.1). We first obtain an initial decomposition of the rule into some sub-patterns, then execute the rule on seed programs to derive code-rule mappings that link each sub-pattern to the actual code it matches. We then leverage a refinement loop on the initial decomposition. The refinement merges the sub-patterns that consistently map to tightly coupled code blocks to preserve context, and splits those mapping spans multiple blocks to prevent overfitting.

Challenge 2: Retrieving Semantically Relevant but Structurally Diverse Code. For each decomposed sub-pattern, we need to retrieve real-world code that is functionally equivalent yet implemented differently. A natural starting point is embedding-based retrieval, which retrieves code snippets through vector similarity. However, this approach encounters two limitations: (1) indexing repositories carries prohibitive computational cost, and (2) similarity search is optimized to retrieve the closest neighbors, yielding homogeneous instead of diverse results.

Solution. We solve this challenge through a combination of *anchor feature queries* and *GitHub Code Search*. GitHub Code Search [11] solves the scale problem: it indexes all public repositories and supports fast keyword queries without pre-computing embeddings. Meanwhile, *anchor feature queries* balance the semantic and diversity. We extract security-critical surface features (see Section 4.2) from the mapped code snippet, combine them into multiple short conjunctive queries, and use a generate-evaluate loop to discard queries that are too broad or too specific. Short queries anchor retrieval in the correct functional domain without fingerprinting the exact code, leaving room for structural variation in the returned snippets. For example, the snippets in Figure 1(c) are retrieved via "`tempfile AND create AND path`".

Challenge 3: Providing an Effective Oracle for Synthesized Variants. The final challenge is determining whether each synthesized variant constitutes a genuine defect. Static oracles such as metamorphic relations assume label-preserving transformations, which do not hold when we splice real-world code into an existing test case. Dynamic oracles such as runtime sanitizers are also unsuitable. They require fully executable code with resolved dependencies, which is impractical for the diverse, multi-library vulnerability patterns we target. More fundamentally, dynamic oracles primarily target memory safety issues (e.g., buffer overflows, use-after-free) and concurrency bugs, whereas our focus includes a broader range of high-level semantic flaws such as business logic bypasses, authentication omissions, and injection vulnerabilities that do not necessarily manifest as runtime crashes.

Solution. We adopt a *rule-specific LLM-as-a-Judge* as a *differential oracle*. While LLMs are imperfect for general vulnerability detection [41, 44], they can still serve as effective differential oracles because our goal is not absolute correctness but systematic

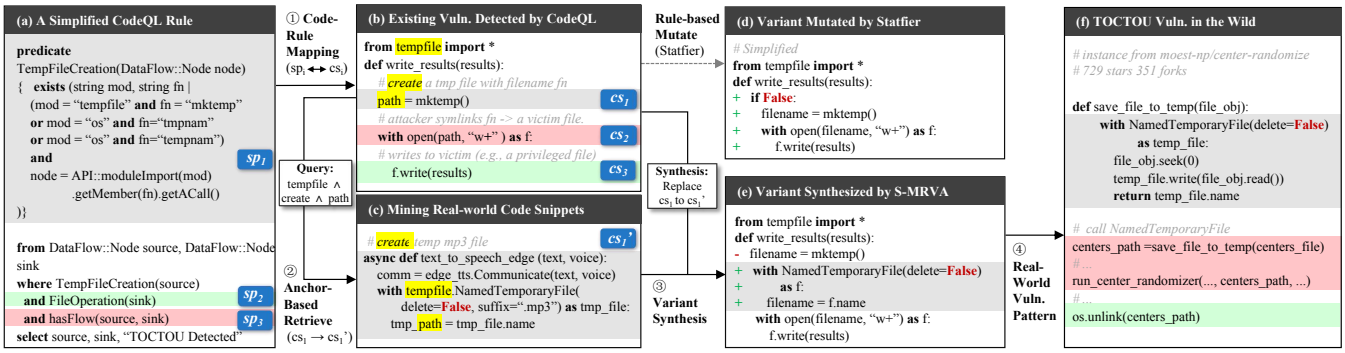


Figure 1: A Motivation Example: Statfier and S-MRVA generated variants for a TOCTOU vulnerability. The highlighted colored blocks map CodeQL sub-patterns (i.e., sp_i) to concrete vulnerable and retrieved code snippets (i.e., cs_i and cs_i', respectively).

divergence from the SAST tool's output. Notably, the synthesis and judgment steps run in separate, memory-isolated LLM sessions with independent task formulations, reducing the risk of correlated errors. Moreover, the variants are compact, self-contained fragments, where LLMs achieve substantially higher accuracy than on whole-repository analysis [44]. Our experiments confirm a false-report rate under 20% (see Section 6.1), competitive with or lower than traditional oracles [25, 42].

3 Problem Formalization

Input and Ground Truth. The primary inputs to our framework are a target SAST rule $\mathcal{R}$ and a set of seed test cases $\mathcal{T}$ known to trigger the vulnerability that $\mathcal{R}$ targets. The SAST rule acts as a binary classifier over individual test cases: for any $t \in \mathcal{T}$, $\mathcal{R}(t) \rightarrow \{\texttt{True}, \texttt{False}\}$, where `True` indicates that $\mathcal{R}$ detects the vulnerability in t. By construction, every seed case satisfies $\mathcal{R}(t) = \texttt{True}$. We define an oracle function $O(t) \rightarrow \{\texttt{True}, \texttt{False}\}$ that provides a ground-truth security judgment.

Rule Structure. A sub-pattern sp is a logical constituent of a SAST rule, denoted $sp \in \mathcal{R}$. Like the full rule, each sp acts as a classifier over code snippets: $sp(cs) \rightarrow \{\texttt{True}, \texttt{False}\}$. A code snippet cs is a contiguous fragment of source code within a test case t.

Code-Rule Mapping. The mapping $\mathcal{M}$ associates each sub-pattern sp with the precise code snippet(s) it matches during analysis of a test case t. It collects these associations across all seeds:

$$\mathcal{M}(\mathcal{R}, \mathcal{T}) = \{\langle sp, cs\rangle \mid sp \in \mathcal{R},\ t \in \mathcal{T},\ cs \in t,\ sp(cs) = \texttt{True}\}.$$

Intuitively, $\mathcal{M}$ tells us *which sub-pattern matched which concrete code* in the programs under analysis.

Output and Defect Definitions. A *variant* $\hat{t}$ is a test case derived from a seed $t \in \mathcal{T}$ by synthesizing real-world code into the seed. S-MRVA output is a set of defect-revealing variants $\mathcal{D} = \{\hat{t}_1, \hat{t}_2, \ldots, \hat{t}_n\}$, each exposing either a false positive or a false negative. A false positive (FP) occurs when $\mathcal{R}$ erroneously flags a secure variant, i.e., $\text{FP}(\hat{t}) \leftrightarrow \mathcal{R}(\hat{t}) \wedge \neg O(\hat{t})$. A false negative (FN) occurs when $\mathcal{R}$ fails to detect a vulnerable variant, i.e., $\text{FN}(\hat{t}) \leftrightarrow \neg\mathcal{R}(\hat{t}) \wedge O(\hat{t})$.

4 Approach

Figure 2 shows the overview of our approach, S-MRVA. Given a SAST rule $\mathcal{R}$ and a seed corpus $\mathcal{T}$ collected from the regression test suites of SAST, S-MRVA processes them through a multi-stage pipeline to produce defect-revealing variants $\mathcal{D}$.

Preliminary Analysis. Before the main processing stages, S-MRVA takes the target rule $\mathcal{R}$ and its seed corpus $\mathcal{T}$ as input, and interprets the detection scope of $\mathcal{R}$ by analyzing its metadata and the coverage of existing variants in $\mathcal{T}$. Specifically, we prompt the LLM to answer two questions with the provided context: (1) what vulnerability this rule is expected to match, and (2) what variant patterns are already covered by the existing seed test cases. The answer (i.e., a natural-language specification) will be carried throughout the subsequent LLM queries as system prompts.

Semantic Decomposition (§4.1). This stage consumes the target rule $\mathcal{R}$ and the seed corpus $\mathcal{T}$. It first decomposes the SAST rule into fine-grained sub-patterns, and constructs the code-to-rule mappings. Guided by the mapping, S-MRVA then iteratively refines the decomposition, producing a set of refined sub-patterns $\{sp_1, \ldots, sp_k\}$ along with their corresponding mappings $\mathcal{M}_i$ as output for downstream analysis.

Retrieval and Variant Synthesis (§4.2). Taking the decomposed sub-patterns $\{sp_i\}$, their mappings $\mathcal{M}_i$, and the original seeds $\mathcal{T}$ as input, this stage constructs query strings from anchor features in the mapped code snippets and retrieves real-world code that exhibits alternative or mitigating patterns. These snippets are carefully integrated into the seed to synthesize candidate variants $\{\hat{t}\}$, which serve as the output passed to the validation stage.

Validation and Root-Cause Clustering (§4.3). Finally, this stage receives the candidate variants $\{\hat{t}\}$ as input, and validates these candidates through differential testing, comparing the SAST rule's output against an LLM-based judgment. The synthesis and judgment steps operate under isolated contexts, reducing the risk of correlated errors. Verified defects are then grouped by root cause and compiled into a structured report, forming the final output $\mathcal{D}$.

Each major stage comprises multiple fine-grained semantic (processed with LLM) and deterministic (implemented with deterministic code) sub-steps, and is guarded by deterministic validators before moving to the next step for reliability. The following subsections introduce these steps, covering the design rationale, the LLM's role and task, and other important details.

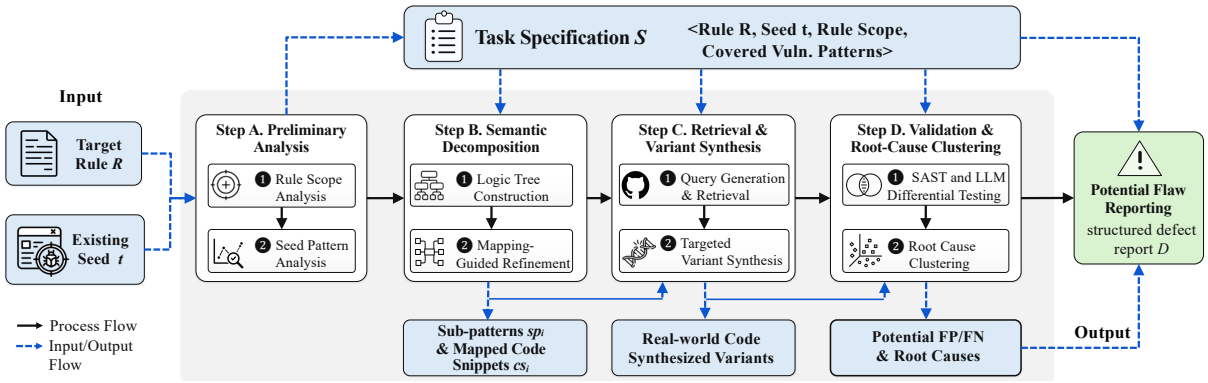


Figure 2: Overview Architecture of S-MRVA.

4.1 Semantic Decomposition

The task of rule decomposition is to divide an SAST rule $\mathcal{R}$ into a set of manipulable sub-patterns. As discussed in Section 2, achieving suitable granularity is non-trivial. To address this, we introduce a helper structure called the *logic tree*, which serves two foundational purposes. First, SAST analyzers use heterogeneous rule formats (e.g., CodeQL queries, Semgrep YAML rules, and native checker implementations), so a unified, codebase-agnostic intermediate representation is necessary. Second, a compositional tree structure explicitly encodes the conjunctions, disjunctions, and negations that govern the detection logic of the rule, thereby supporting deterministic downstream synthesis.

We define a logic tree as an abstraction that captures both the sub-patterns of a SAST rule and their logical relationships:

$$Leaf ::= \langle d, sp \rangle,$$
$$Node ::= Leaf \mid (Node \wedge Node) \mid (Node \vee Node) \mid \neg Node.$$

Here, d denotes a natural-language description of the sub-pattern's semantics, and sp denotes the sub-pattern itself in its concrete implementation (e.g., YAML, QL query, etc.).

4.1.1 Logic Tree Construction. To achieve an analyzer-agnostic logic tree construction, S-MRVA leverages the LLM as a general parser for SAST rules in different formats. The logic tree schema and the rule implementation are provided to the LLM, which is instructed to review the implementation from top to bottom and construct a line-by-line abstraction following the schema. This step produces $\mathcal{L}_{init}$, where each leaf carries a semantic description d and a linked, analyzer-specific implementation sp. For example, the first AND-branch of the `TempFileCreation` predicate in Figure 1(a) is initially parsed as: $\langle$`filename should be "mktemp"`$, sp_1\rangle \wedge \langle$`modulename should be "tempfile"`$, sp_2\rangle$.

4.1.2 Mapping-Guided Refinement. At this point, the decomposition is typically too fine-grained. For a complex SAST rule, $\mathcal{L}_{init}$ may contain dozens of leaves, which are unwieldy for downstream synthesis. S-MRVA therefore initiates a refinement loop on $\mathcal{L}_{init}$, guided by code-to-rule mappings. Specifically, there are two steps: (i) deterministically running the analyzer on each seed t_i to collect the matching context. (ii) using an LLM to construct $\mathcal{M}_i$, which links each semantic description $d \in \mathcal{L}_{init}$ to the triggering code snippets in cs_i. Based on these mappings, the LLM acts as a decider, determining merge/split actions across nodes in $\mathcal{L}$ constrained by three policies in the prompt as shown below. Note that LLMs only make decisions. S-MRVA then applies precise and deterministic restructuring operations with code to ensure valid formats.

AND-merge policy: For two sibling nodes under an AND clause, if the mappings across seeds show that both nodes are always activated together and consistently map to a single tightly coupled target, the system merges them into one node (e.g., in Figure 1(a), `mod=tempfile` and `fn=mktemp` always co-occur in the same API call and merge into `tempfile.mktemp`). This removes artificial fragmentation of what is semantically one unit.

OR-merge policy: For two sibling leaves under an OR clause, if their mappings fall within the same code block and play the same semantic role, the system merges them into one alternative node (e.g., `tempfile.mktemp`, `os.tmpnam`, and `os.tempnam` are merged into one `TempFileCreation` node as they are alternative implementations of one operation). This preserves a single semantic choice point instead of many isolated leaves.

Split policy: If one node maps to code that spans across code blocks or even functions, the node is overly complex for mutation. In this case, the system splits it into smaller, more focused nodes (e.g., splitting the inter-procedural `hasFlow` predicate in Figure 1(a) from adjacent nodes).

To control the granularity of the refined tree, S-MRVA enforces a *bound rule* using two global thresholds, an upper bound B_{high} (discussed in Section 6.6) and a lower bound B_{low}. The refinement loop runs iteratively. After each turn, the validator checks the current leaf count against the bounds and directs the LLM to prioritize merges when the count exceeds B_{high}, or splits when it falls below B_{low}. The loop terminates once the leaf count falls within the bound interval, and the leaves of $\mathcal{L}$ are output as the final decomposed sub-patterns. In practice, we fix $B_{low} = 2$ as a quality gate to ensure that the decomposition works (i.e., yields at least two sub-patterns).

4.2 Retrieval and Variant Synthesis

After decomposing a SAST rule into semantic sub-patterns, we conduct a targeted retrieval and variant synthesis process for each sub-pattern. For a given sub-pattern sp_i, its associated seed t, and the mapped code snippet cs_i within that seed, we generate GitHub

Code Search query strings $Q = \{q_1, q_2, \ldots, q_n\}$ to retrieve relevant and diverse code snippets from real-world repositories. After filtering, each retrieved snippet cs'_i is used to synthesize a variant $\hat{t}$ by replacing cs_i in t with cs'_i and re-wiring the surrounding data flow.

4.2.1 Query Generation & Retrieval. A naive strategy of randomly combining terms extracted from cs into search queries is unlikely to succeed since short combinations will match vast amounts of unrelated code while long combinations collapse to near-exact copies of cs. Neither produces the semantically related but structurally diverse code needed for effective variant synthesis.

To address this problem, we ground each query in *anchor features* extracted from the target sub-pattern sp (API calls, framework patterns, argument values, and domain identifiers) and their *plausible alternatives* (i.e., functionally equivalent constructs implemented differently, e.g., `os.tempnam` vs. `tempfile.mktemp`). Including alternatives exposes the search to diverse implementation styles that may bypass the rule, whereas using only the exact anchors would retrieve near-duplicates of the original snippet. Specifically, S-MRVA prompts the LLM to compose conjunctive queries from these anchors and alternatives. To avoid the overly-specific extreme, we cap each query at three features, inspired by the design of Google Code Search.[6] This small, fixed-size conjunction anchors retrieval in the correct functional domain without fingerprinting cs, leaving room for structural variation in the returned snippets. We also generate multiple independent queries with different feature combinations to increase the diversity of retrieval.

To further ensure the quality of generated queries, we adopt an evaluator-optimizer loop[7] that first retrieves the top 5 candidates for each generated query. A query is discarded if it returns too few results (overly specific) or if the evaluator judges the 5 candidates as irrelevant to the target sub-pattern (overly broad). Discarded queries are fed back into the loop, where the LLM relaxes or narrows them accordingly. Only queries that pass this evaluation will proceed to the full search. For example, given cs_1 in Figure 1(b), the query "create ∧ path ∧ file" loses the semantics of temporary file creation and is dropped, whereas "tempfile ∧ path ∧ create" is retained.

After obtaining the query set Q, the system performs a full code search and collects a raw set of retrieved snippets $\mathcal{S}$. Formally, each result in $\mathcal{S}$ is a tuple $\langle org, repo, file, cs' \rangle$, where org, $repo$, $file$, and cs' denote the organization name, repository name, file path, and matched code content, respectively. The snippet cs' includes a maximum window of ten lines around the exact match to preserve the surrounding context [32].

Public repositories frequently contain copies, forks, and reused examples, so the raw set is noisy and repetitive. To address this, S-MRVA applies a filtering step to remove duplicates and similar cases: (1) results with exactly the same cs', (2) results from the same $repo$ but different org (i.e., forks), and (3) results matched in the same $file$. After that, we further remove near-duplicate code by embedding each snippet and clustering with a cosine similarity threshold τ (discussed in RQ6), keeping one representative per cluster, yielding a cleaner candidate pool $\mathcal{S}'$.

To further improve efficiency, we introduce a semantic step in which the LLM acts as a judge, tasked with evaluating whether a given candidate snippet cs' bypasses the target sub-pattern sp. The LLM assesses the candidate against two criteria: 1) *Alternative Patterns:* cs' performs the same function as cs but uses a different implementation style. This snippet may bypass the sub-pattern sp and trigger a *false negative* in the overall rule. 2) *Mitigation Patterns:* cs' uses the same implementation as cs but includes additional validation or mitigation logic in its context. If the SAST tool fails to recognize the mitigation and still reports the code as vulnerable, it will trigger a *false positive*. S-MRVA forwards the candidates matching either criterion as $\hat{\mathcal{S}}$ to the synthesis step.

4.2.2 Targeted Variant Synthesis. Once the filtered retrieval results $\hat{\mathcal{S}}$ are obtained, S-MRVA splices them back into the seed t by replacing the mapped code snippet cs with the retrieved implementation cs' to synthesize a new variant $\hat{t}$. However, direct replacement can break the semantics of the test case. For instance, it may introduce unrelated statements, disrupt data-flow dependencies, or break variable bindings. Such issues can lead to runtime failures or cause the SAST tool to misclassify the mutated program.

To address this, S-MRVA employs a three-step semantic workflow, where the LLM acts as a synthesizer, tasked with integrating the retrieved candidate snippet cs' into the original seed t to produce a plausible variant $\hat{t}$. First, the LLM performs semantic alignment, eliminating irrelevant statements from cs' (e.g., unrelated logging, error handling, and local bookkeeping that do not belong to the anchor operation or its mitigation/alternative pattern) while preserving the risky structure (i.e., the alternative or mitigation pattern) against the sub-pattern. Second, it keeps all code outside the mapped region unchanged, substitutes only the mapped snippet with the trimmed cs', and reconnects the local data flow using variables and values that already exist in the original code. Third, it applies minimal edits to make $\hat{t}$ self-contained, for instance, adding package imports and initialization for the newly introduced code. This structured approach enables effective synthesis while reducing noise inherited from the raw retrieval results.

To ensure that the SAST analyzers can actually execute on the synthesized variants rather than failing before any analysis takes place, we validate each $\hat{t}$ against the corresponding tool. For analyzers that require compilation (e.g., CodeQL), we verify that compilation succeeds and discard variants that fail to compile. For source-based analyzers (e.g., Semgrep), we run the analyzer directly and discard variants that produce parser errors. This filtering prevents functional false positives caused by unanalyzable inputs.

4.3 Validation and Root-Cause Clustering

4.3.1 SAST and LLM Differential Testing. We validate variants through differential testing. Specifically, for each synthesized variant $\hat{t}$, we compare the SAST ruling $\mathcal{R}(\hat{t})$ against the judgment $O(\hat{t})$ of an LLM-as-a-judge oracle. A discrepancy ($\mathcal{R} \neq O$) flags $\hat{t}$ as a ***potential*** flaw. To improve the reliability of the LLM oracle, we construct a rule-specific agent for each SAST rule $\mathcal{R}$ by materializing the preliminary analysis (detection scope and already-covered variant patterns) into its system prompt. More importantly, we *isolate* the LLM agent for judgment from the synthesis stages. It is invoked in a separate LLM context with only $\hat{t}$ and rule scope specification, without seeing the seed code, the retrieved snippet, or any intermediate synthesis decisions. Our experiments confirm

[6] https://swtch.com/~rsc/regexp/regexp4.html
[7] https://platform.claude.com/cookbook/patterns-agents-evaluator-optimizer

that under this design, the LLM serves as a practical differential oracle, achieving over 80% accuracy (see Section 6.1).

4.3.2 Root Cause Clustering. Since S-MRVA synthesizes variants from real-world bypass patterns, the pipeline is highly effective. In our experiments, roughly one in four variants triggers a discrepancy. Consequently, S-MRVA may produce dozens of potential flaws per rule. Reviewing each independently before submission would impose substantial manual effort. To reduce this burden, we leverage the inherent structure of the synthesis pipeline: each variant is already associated with a targeted sub-pattern, providing natural coarse categorization. We further introduce a semantic step in which the LLM acts as a *root cause analyzer*, tasked with identifying the implementation-level root cause for the flagged variants. Specifically, the LLM receives the seed-to-variant diff and the sub-pattern implementation as input, and outputs a concise root cause description (e.g., "*Failure to Track Taint Through Asynchronous Callbacks*") and then clusters to consolidate variants that share the same underlying cause. Finally, S-MRVA produces a structured report of flagged variants with reproducible code, defect type, analyzer output, and oracle judgment, all organized by root cause.

While it is effective in practice (i.e., in our experiment, the false report rate is about 18.4%), the LLM-as-a-judge operates as a differential oracle. Specifically, any reported discrepancy may reflect either a genuine SAST rule flaw or an LLM misjudgment. All cases are manually verified by the authors before submission.

5 Evaluation Setup

5.1 RQ Setup

In this section, we evaluate the effectiveness, efficiency, and real-world applicability of our approach (S-MRVA) against state-of-the-art baselines. Specifically, we focus on answering the following seven research questions (RQs):

RQ1 (Effectiveness): How effective is S-MRVA compared to baselines at finding unique flaws?

RQ2 (Security Impact): Can S-MRVA's findings help SAST maintainers fix real security bugs in practice?

RQ3 (Naturalness & Realism): Does S-MRVA generate more natural and realistic code variants compared to existing baselines?

RQ4 (Generalization): How effective is S-MRVA at generating variants for uncommon or rare vulnerability types?

RQ5 (Ablation Study): How do the main designs of our approach contribute to the overall performance?

RQ6 (Parameter Sensitivity): How sensitive is S-MRVA to the decomposition bound and deduplication threshold?

RQ7 (Cost & Extensibility): What are the token/time costs of S-MRVA, how easily does it port to new SAST tools, and how sensitive is it to the choice of LLM backend?

5.2 Dataset and Supported Analyzers

We collect a comprehensive benchmark comprising 220 security rules spanning four prominent SAST tools (i.e., CodeQL, Semgrep, Bandit, and SpotBugs) and five distinct rule sets across three widely used programming languages (i.e., C/C++, Java, and Python), as detailed in Table 1. The selection of these SAST tools is motivated by their popularity, technical diversity, and academic influence. Specifically, Bandit (AST matching), Semgrep (semantic pattern matching), and CodeQL (global data-flow and taint tracking) represent different mainstream program analysis paradigms and are widely used in previous SAST-related work [17, 35, 37]. Additionally, STATFIER and STAAGENT use SpotBugs in their experiments [25, 42], so we also include it to establish a fair and direct basis for comparison.

Table 1: Dataset and Supported Analyzers Used in Evaluation.

Ruleset	Analyzer	Version	# Rules
codeql/py/security (CQL)	CodeQL	2.23.9	48
semgrep/java/security (SGJ)	Semgrep	1.156.0	79
0xdea/security (SGC)	Semgrep	1.156.0	48
bandit/security (BDT)	Bandit	1.9.4	31
spotbugs/security (SBG)	SpotBugs	4.9.3	14
Total	–	–	**220**

To maintain a focused and realistic evaluation scope, we restrict our dataset strictly to ***security-related rules***. For analyzers that support multiple languages, we single out one language subset per tool to avoid redundant evaluations of different language-specific implementations of the same vulnerability categories. Crucially, for each selected rule, we extract its existing officially provided regression test cases from the respective tool's repository to serve as the initial seed snippets for variant generation.

5.3 Baselines

We compare against four representative baselines:

STATFIER [42]: A state-of-the-art, heuristic-driven metamorphic testing framework that applies semantic-preserving syntax mutations to expose FPs/FNs in static analyzers.

STAAGENT [25]: The most recent, closest SAST testing approach, extending Statfier with LLM-based mutation and execution-tracing oracle. We include it as a representative LLM baseline.

SA_BUGS [14]: A fault-injection-based defect detector for C/C++ static analyzers. It injects true-positive code snippets into auto-generated C code (via Csmith [40]) to verify whether static analyzers can detect the injected defects with a dynamic oracle.

MRVA [12]: A popular mining-based variant analysis pipeline natively integrated into CodeQL. MRVA scales variant detection across thousands of real-world repositories to query and identify organic occurrences of vulnerability patterns. We include MRVA not as a competing oracle-based method, but to establish the ceiling of pure repository mining: it excels at scale for prevalent patterns but is structurally blind to false negatives and rare vulnerabilities.

Fair comparison across all rule sets is hindered by hard constraints in the core pipeline of baselines. Specifically, Statfier's mutator is hard-coded to Java AST nodes, SA_bugs depends on the C-only Csmith generator, StaAgent's dynamic validation is tightly coupled to Java, and MRVA is accessible only through GitHub Actions and supports only CodeQL. We therefore adapted baselines to the largest feasible subset of our rules *without modifying their core mechanisms*. Particularly, Statfier and StaAgent were extended from SpotBugs (SBG) to also support the Semgrep Java rule set (SGJ), and SA_bugs was extended to the Semgrep C rule set (SGC).

For every adaptation, we verified that the pipeline produced comparable output volume and finding counts on both the originally supported target and the new target.

5.4 Metrics

A test case is classified as a *potential FP/FN* (Pot.) if its SAST analysis result contradicts the corresponding oracle judgment. To reduce redundancy, all tools (including S-MRVA and baselines) perform automatic deduplication on their raw outputs. The resulting set of distinct cases is denoted as *reported FP/FNs* (Rep.). Theoretically, both our approach and the baselines may produce false reports at this stage. We therefore incorporate a manual check to identify *unique defects* (Uniq.) following the root-cause classification policy established in prior work [14, 42]. Based on these definitions, we adopt the following metrics to evaluate the effectiveness, security impact, and realism of S-MRVA against the baselines.

$\mathbf{ER_{pf}}$ (Potential Flaw Rate) measures the proportion of generated variants that trigger a potential FP/FN:

$$ER_{pf} = \frac{|\text{Pot.}|}{|\text{Variants}|}$$

A higher ER_{pf} indicates that the approach is more capable of generating variants likely to expose rule flaws.

$\mathbf{ER_{ud}}$ (Unique Defect Rate) measures the proportion of reported cases that correspond to manually verified, distinct defects:

$$ER_{ud} = \frac{|\text{Uniq.}|}{|\text{Rep.}|}$$

A higher ER_{ud} suggests that the tool produces actionable, non-redundant bug reports with fewer false alarms.

Defect Coverage (Generalization) measures the proportion of target SAST rules for which at least one defect is successfully identified. A higher coverage indicates a stronger capability to reliably expose defects across diverse vulnerability categories.

We measure variant realism along four dimensions. Provenance from real GitHub repositories (discussed in Section 4) establishes an intrinsic grounding in actual developer practice, while maintainer-confirmed production impact (validated in RQ2 through community-reported findings and feedback) serves as the extrinsic validator. As these two signals are inherently qualitative, we further introduce two metrics that enable quantitative comparison. Following prior work on code naturalness and diversity, we adopt:

Perplexity (Naturalness): Measures code naturalness via the average negative log-likelihood of mutant tokens ($x_1 \dots x_N$) using causal code LLMs. Lower perplexity denotes that the code exhibits higher resemblance to human ones [18, 19]:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(x_i \mid x_{<i})\right) \tag{1}$$

Feature Space Dispersion (Diversity): Assesses diversity via the average pairwise cosine distance among mutant embeddings ($e_i, e_j \in M$). Values closer to the GitHub baseline indicate a more realistic variance:

$$\text{Dispersion} = \frac{2}{|M|(|M|-1)}\sum_{1\le i<j\le |M|}\left(1-\frac{e_i \cdot e_j}{\|e_i\|\|e_j\|}\right) \tag{2}$$

$$\text{Gap} = \text{Dispersion}_{tool} - \text{Dispersion}_{github}$$

5.5 Implementation and Configurations

We implement a prototype of S-MRVA. S-MRVA is implemented as a Python orchestration framework built with LangGraph, integrating the GitHub Code Search (via GitHub REST API) for real-world snippet retrieval. We set the LLM temperature to zero for more reproducible outputs following existing studies [29, 38]. We use a local embedding model (CodeT5p) to cluster raw code snippets retrieved from GitHub and select one representative case per cluster, setting the cosine similarity threshold τ to 0.6 to ensure compact yet representative clusters without excessive fragmentation (see discussion in Section 6.6). We set $B_{high} = 5$ to balance the overall performance and computational overhead (see analysis in Section 6.6). We use *qwen3.5-plus-2026-04-20* from the official API provider as the primary LLM backend for cost efficiency and strong code understanding capabilities (we evaluate S-MRVA with another three LLMs in RQ7). We set up a timeout of 24 hours for S-MRVA and all the baselines. For all baselines, we use the configurations as recommended in their guidelines. For mutation-based baselines (i.e., Statfier and StaAgent), we start with the same seeds from the existing test suite. All our experiments are executed on a server using Ubuntu 24.04 with an AMD EPYC 64-core CPU and 512GB RAM, and two RTX 3080Ti GPUs.

6 Evaluation Results

6.1 RQ1: Overall Effectiveness

As shown in Table 2, S-MRVA demonstrates strong capability in identifying flaws across all evaluated analyzers. From 220 target rules, S-MRVA generated 6,206 variants and triggered 1,735 potential FPs/FNs, yielding an average ER_{pf} of 27.96%. Automatic clustering produced 561 root-cause groups. After manual review (Section 5.4), we obtained ***458 unique defects***, with a consistently high ER_{ud} (averaging 81.6%) and broad coverage (214 out of 220 rules, 97.27%).

6.1.1 Comparison with Baselines. Table 3 compares S-MRVA against the four baselines. S-MRVA substantially outperforms all of them in both unique defects found and rule coverage. Specifically: 1) *Java (SBG & SGJ):* S-MRVA achieves 100% rule coverage with 40 and 157 unique defects, versus STATFIER's 1 and 24 (21–28% coverage) and far lower ER_{ud} (33% and 24% vs. S-MRVA's 82–84%). STAAGENT found *zero* defects under our setup: its dynamic oracle requires executable mutants, but many security test cases are inherently non-runnable (e.g., snippets connecting to external databases or web services), causing its pipeline to abort. 2) *C/C++ (SGC):* S-MRVA discovered 101 unique defects (ER_{ud} = 82.11%, 97.92% rule coverage), while SA_BUGS, with Csmith-based template injection, yielded 5 unique defects (ER_{ud} = 14.29%). 3) *Python (CodeQL):* S-MRVA pinpointed 99 unique defects via targeted generation. MRVA retrieved 2,836 candidates by mining the top 100 Python repositories, but lacks a

Table 2: Overall Effectiveness and Defect Coverage of S-MRVA (ER_{pf} = Pot. FP/FN/Mutants; ER_{ud} = Unique/Rep. FP/FN).

Ruleset	# Rules	Internal			Rep. FP/FN	Manual Verified		Defect Cov.
		Mutants	Pot. FP/FN	ER_{pf}		Uniq.	ER_{ud}	
CQL	48	1,329	364	27.39%	119	99	83.19%	46 (95.83%)
SBG	14	528	207	39.20%	49	40	81.63%	14 (100.00%)
BDT	31	865	234	27.05%	82	61	74.39%	28 (90.32%)
SGJ	79	2,146	554	25.82%	188	157	83.51%	79 (100.00%)
SGC	48	1,338	376	28.10%	123	101	82.11%	47 (97.92%)
Total	**220**	**6,206**	**1,735**	**27.96%**	**561**	**458**	**81.64%**	**214 (97.27%)**

Table 3: Comparison against Baselines. For Uniq., numbers in parentheses indicate duplicate reports within `Rep.` that were excluded during manual inspection. "N/A" indicates the data cannot be obtained by the baselines.

Ruleset	Approach	Rep.	Uniq.	ER_{ud}	Defect Cov.
SGJ	**S-MRVA**	**188**	**157 (0)**	**83.51%**	**79 / 79 (100.00%)**
	STATFIER	102	24 (76)	23.53%	22 / 79 (27.85%)
	STAAGENT	0	0 (0)	0.00%	0 / 79 (0.00%)
SBG	**S-MRVA**	**49**	**40 (0)**	**81.63%**	**14 / 14 (100.00%)**
	STATFIER	3	1 (0)	33.33%	3 / 14 (21.43%)
	STAAGENT	0	0 (0)	0.00%	0 / 14 (0.00%)
SGC	**S-MRVA**	**123**	**101 (0)**	**82.11%**	**47 / 48 (97.92%)**
	SA_BUGS	35	5 (30)	14.29%	N/A
CQL	**S-MRVA**	119	**99 (0)**	**83.19%**	**46 / 48 (95.83%)**
	MRVA	**2,836**	N/A	N/A	N/A

ground-truth oracle, making manual triage prohibitive. Moreover, mining-based approaches inherently cannot detect false negatives.

6.1.2 Effectiveness of LLM-as-a-Judge. Both Statfier and SA_bugs show a steep drop from their reported defects to unique, verified defects (i.e., a low ER_{ud}). The primary reason is that both systems rely on heuristic-based mutation strategies with limited variation, which still produce a large number of duplicate reports after the deduplication method in their approaches (i.e., program reduction [14]). We use the transformation sequences to further deduplicate the results from Statifer since there are too many noise [42]. As shown by the numbers in parentheses, Statfier has 76 duplicates out of 102 reports in the SGJ ruleset (and 2 false reports), and SA_bugs has 30 duplicates out of 35 in the SGC ruleset. These duplicates would require substantial manual deduplication effort before any report can be acted upon. In contrast, S-MRVA integrates a clustering mechanism that leverages both the targeted sub-pattern and the identified root cause to consolidate reports automatically. The mechanism is effective for report deduplication (no duplicate reports were observed during manual inspection). Consequently, the ER_{ud} gap for S-MRVA is primarily caused by the LLM-as-a-Judge differential oracle, at a rate of 18.4% overall. This false-report rate is comparable to that of Statfier (16%) and substantially lower than StaAgent (over 50%), as presented in their papers [25, 42].

In summary, the LLM-based verification introduces a modest false-positive overhead (18.4%), comparable to Statfier and substantially lower than StaAgent. More importantly, it produces fewer duplicate reports than the baselines with root-cause clustering. In addition, the semantic oracle enables S-MRVA to test with more diverse mutants, leading to substantially more unique, actionable defects than prior approaches. These results confirm that LLM-as-a-Judge is both practically feasible and effective for rule testing.

6.2 RQ2: Security Impact

6.2.1 Flaws Reported to SAST Maintainers. While reporting every case indiscriminately would overwhelm maintainers, we prioritize them according to the severity and confidence scores defined in the rule metadata, as assigned by the maintainers. We additionally filter by fix feasibility, withholding 15 findings that represent real-world scenarios but are unlikely to receive fixes: prohibitive performance overhead, missing taint support, or explicit tool scope boundaries. For each unique defect, we construct a minimal PoC and a detailed description of the flaw.

As shown in Table 4, we have filed 89 unique defects across 43 reports, covering the top 20% of prioritized rules (defects per rule are consolidated and reported in one issue). ***Among the 21 reports that have received a response (labels, pull requests, or comments), 20 have been confirmed (95.2%).***, and the remainder are pending. The high pending rate is due to limited maintainer bandwidth and the heavy backlog of open issues in these projects. Moreover, community-maintained rulesets (e.g., Semgrep, SpotBugs) lack dedicated triage personnel. Consequently, the feedback duration varies from days to months.

Encouragingly, we received positive feedback from the maintainers, which strongly confirms the practical usefulness of our approach. Specifically, with our reported issues, CodeQL maintainers refined their rule and evaluated it on the top 1,000 most popular Python projects, and ***found three new vulnerabilities and reduced three false alarms***.[8]

> *"The baseline run had 170 alerts, and the improved query has 212 ... Deduplicating and comparing the results, we lose 3 FPs, and gain 3 TPs, with everything else remaining essentially the same."*

We also received feedback from the fix commit of the Semgrep team that our submitted flaw has finally ***affected 18 crypto rules in their ruleset and led to a total update***.[9]

> "... updated 18 rules across Java and Kotlin to use (?i) case-insensitive matching via metavariable-regex or inline regex patterns."

[8] https://github.com/github/codeql/pull/21590#issuecomment-4198864487
[9] https://github.com/semgrep/semgrep-rules/pull/3798

Table 4: Issues Reported. "#Num." indicates the number of unique defects contained in each report. "BL" indicates whether the defects can also be detected by any baselines.

Tool	Rule	#Num.	LINK	STATUS	BL
BDT	B105	3	I1383	CONFIRM	×
BDT	B103	1	I1390	PENDING	×
BDT	B202	1	I1392	PENDING	×
BDT	B501	2	I1394	PENDING	×
BDT	B104	1	I1395	CONFIRM	×
BDT	B506	2	I1397	CONFIRM	×
CQL	insecure-protocol	2	I21666	PENDING	×
CQL	hardcoded-credentials	3	I21620	CONFIRM*	×
CQL	flask-debug	2	I21616	PENDING	×
CQL	bind-to-all-interface	3	I21582	FIXED	×
CQL	insecure-temp***	1	I21669	PENDING	×
CQL	insecure-cookies	2	I21647	CONFIRM	×
CQL	clear-text-***	2	I21595	CONFIRM*	×
CQL	tarslip	2	I21712	CONFIRM	×
CQL	path-injection	1	I21711	WON'T FIX	×
CQL	ldap-injection	1	I21738	PENDING	×
CQL	code-injection	1	I21735	PENDING	×
SBG	DMI_CONSTANT_***	4	I3990	PENDING	×
SBG	EI_EXPOSE_REP	2	I4003	FIXED	×
SBG	MS_EXPOSE_BUF	2	I4007	FIXED	×
SGJ	ecb-cipher	3	I11662	PENDING	×
SGJ	expat-xxe	1	I3724	PENDING	×
SGJ	java-jwt-decode-***	1	I11410	CONFIRM	×
SGJ	engine-bitwise	1	I11407	FIXED	×
SGJ	java/crypto/*	18	I11408	FIXED	×
SGJ	weak-random	1	I3732	PENDING	×
SGJ	engine-inherit	1	I11435	CONFIRM	×
SGJ	server-dangerous-***	1	I3737	PENDING	×
SGJ	command-injection-***	1	I3763	PENDING	×
SGJ	engine-call-chain	1	I11545	FIXED	×
SGJ	jjwt-none-alg	1	I3759	PENDING	×
SGJ	documentbuilder***	3	I3831	PENDING	×
SGJ	anonymous-ldap-bind	1	I3758	PENDING	×
SGJ	hibernate-sqli	2	I3817	PENDING	×
SGJ	overly-permissive-***	3	I3818	PENDING	×
SGJ	tainted-url-host	3	I3819	PENDING	×
SGJ	engine-array-type	1	I11483	CONFIRM	×
SGJ	engine-type-matching	1	I11621	PENDING	×
SGJ	engine-ifdef-parse	1	I11619	CONFIRM	×
SGC	engine-goto-label	1	I11620	CONFIRM	×
SGC	raptor-incorrect-***	3	P17	CONFIRM	×
SGC	incorrect-order-setuid-***	1	I15	FIXED	×
SGC	incorrect-order-setuid-***	1	I18	PENDING	×
ALL	-	89	-	-	-

* Acknowledged by the maintainers but not prioritized for near-term fixes due to performance concerns or product-scope decisions.

6.2.2 Case Study. We present several representative cases as mentioned above, highlighting how those synthesized variants uncover critical blind spots from rule logic to engine correctness.

Case 1: effectiveness in finding wild vulnerabilities. The CodeQL rule for detecting insecure binding missed two common patterns: binding to all interfaces via an environment-variable default (e.g., `os.environ.get('HOST', '0.0.0.0')`, which exposes the service when the variable is unset) and via alternative networking frameworks such as `gevent` and `sanic`.

The box below traces how S-MRVA exposed both patterns. First, decomposition yields three sub-patterns (`imported_framework`, `bind_API_call`, and `hostname_const`) with code-rule mappings to the seed. Then, query generation retrieves two useful idioms (i.e., `cs1`, `cs2`) with the generated queries (i.e., `q1`, `q2`): one using an alternative framework (`gevent`) and another reads the bind address from an environment variable with a fallback to a wildcard address. After that, synthesis splices both retrieved patterns into the seed, producing two variants. Both were flagged as *false negatives* at the verification stage (i.e., the LLM judgment step (with *qwen3.5-plus*) judged them as vulnerable (i.e., the service binds to all interfaces, creating a security risk), whereas CodeQL produced no alert).

```
---------------- Logic Tree (JSON) ---------------------
{ "AND": [
    {"sp1": "imported_framework",
        "impl": "moduleImport(\"socket\")␣..."},
    {"sp2": "bind_API_call",
        "impl": "getMember(\"bind\").getACall()␣..."},
    {"sp3": "hostname_const",
        "impl": "hostname␣in␣{'0.0.0.0','','::','::0'}"}]}
---------------- Code-rule Mapping ---------------------
sp1 -> from socket import socket
sp2 -> s = socket(); s.bind(...)
sp3 -> host = '0.0.0.0'; port = 8080
------- Queries & Retrieved Code Snippets --------------
# q1 : "import AND socket AND sock"
# cs1: gevent/gevent: src/gevent/server.py
import gsocket
sock = GeventSocket(family=family)
sock.bind(address)
# q2 : "host AND 0.0.0.0 AND lang:py"
# cs2: BiznetGIO/RESTKnot: api/gunicorn.conf.py
host = os.environ.get("APP_HOST", "0.0.0.0")
port = os.environ.get("APP_PORT", "8000")
bind = f"{host}:{port}"
---------- Synthesized Variant (DIFF) ------------------
# FN1 (synthesize cs1)
- import socket
+ import gsocket
# FN2 (synthesize cs2)
- host = '0.0.0.0'
+ host = os.environ.get('APP_HOST', '0.0.0.0')
s.bind((host, 8080))
```

After manual confirmation, we reported both of them, and the maintainer confirmed the gap and deployed a fix.[10] The fixed rule later detected three new real-world vulnerabilities in popular open-source projects, while reducing false positives (as reported by the CodeQL team).[11]

Case 2: exposing widely affected flaw. In the second case, involving Semgrep, we found that numerous Java/Kotlin cryptographic rules strictly checked string literals based on hardcoded casing (e.g., `"Blowfish"`). This directly ignores the Java Security API standard and common coding practices, which specify that algorithm names are case-insensitive.

```
----------------Real-World Code Snippet--------------
mCipherOut = Cipher.getInstance("blowfish");
mCipherIn = Cipher.getInstance("blowfish");
if ((mCipherOut == null) || (mCipherIn == null)) {
    throw new IOException("Can't␣load␣blowfish")}
-------------Synthesized Variant(DIFF)----------------
- KeyGenerator.getInstance("Blowfish");
+ KeyGenerator.getInstance("blowfish"); // FN
```

This single semantic disconnect systematically affected dozens of individual rules targeting weak algorithms (e.g., MD5, SHA1, DES,

[10] https://github.com/github/codeql/pull/21590
[11] https://github.com/github/codeql/pull/21590#issuecomment-4198864487

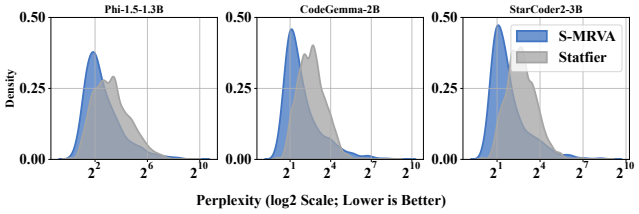


Figure 3: Kernel Density Estimation of Code Perplexity for S-MRVA and Statfier Variants across three Language Models.

Table 5: Feature Space Dispersion Gap (smaller is better).

Embedding Model	Gap: S-MRVA	Gap: Statfier	*P*-Value
CodeT5p	**0.2377**	0.2974	1.87×10^{-8}
CodeBERT	**0.0072**	0.0082	4.21×10^{-6}
EmbeddingGemma	**0.2357**	0.2999	2.07×10^{-7}

ECB padding), causing trivial yet widespread blind spots across their active security rulesets. *Following our report,* ***18*** *affected rules were updated immediately.*[12]

Case 3: detecting engine flaw. While S-MRVA is designed to detect flaws in rules, the synthesized variants can also expose foundational engine flaws with real-world idioms. S-MRVA synthesized a real-world code snippet using C-style array declaration syntax while testing a rule in Semgrep. However, the engine failed to correctly evaluate the type of `data`, leading to a false negative.[13]

```
----------------Real-World Code Snippet--------------
private void ekskey(byte data[], byte key[]) {
      MessageDigest sha512 = MessageDigest
                             .getInstance("SHA-512");
-------------Synthesized Variant(DIFF)----------------
- void func(byte[] data); //data-> byte[]
+ void func(byte data[]); //data-> byte
```

6.3 RQ3: Naturalness and Diversity

In the motivation example and RQ2, we use case studies to illustrate how variants generated by S-MRVA differ from those produced by baselines. In this section, we complement those case studies with a broader quantitative analysis of naturalness and diversity, comparing S-MRVA against the mutation-based baseline Statfier. Specifically, for each rule, we collect variants generated by S-MRVA and Statfier, and analyze code perplexity and feature-space dispersion to examine whether real-world-grounded synthesis improves the naturalness and diversity of generated variants.

6.3.1 Naturalness in Terms of Perplexity. Naturalness measures how closely a generated code snippet resembles human-written code. We compute *Code Perplexity* [18, 19] of the diff tokens between the seed and each variant using code LLMs. Lower perplexity reflects stronger alignment with the distribution of human-written code, whereas high perplexity reliably identifies unnatural constructs such as Statfier's `always-false` branches. We therefore treat it as a necessary sanity check, not a standalone proof of realism. To mitigate model bias, we employ three causal code LLMs: CodeGemma-2B, Phi-1.5-1.3B, and StarCoder2-3B.

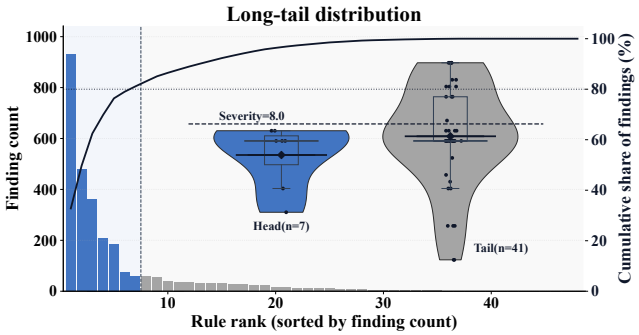


Figure 4: Potential Vulnerabilities Identified by MRVA.

Figure 3 shows the kernel density estimation of perplexity scores for variants generated by S-MRVA and Statfier. A left-shifted distribution denotes lower perplexity (i.e., better naturalness). Across all three models, S-MRVA variants consistently obtain substantially lower perplexity scores, confirming that our approach produces code variants that closely mirror human-authored code.

6.3.2 Feature Space Dispersion. Whereas perplexity captures the naturalness of individual variants, feature space dispersion evaluates how well the overall set of generated variants reflects the structural diversity of real-world code. We quantify dispersion as the average pairwise cosine distance between contextual embeddings extracted via three widely used embedding models: CodeT5p, CodeBERT, and EmbeddingGemma.

We construct a real-world baseline by collecting relevant code snippets from GitHub. For each rule, we retain snippets that use the same APIs or frameworks as the rule's seed. We then embed the variants from S-MRVA and Statfier alongside these real-world snippets and compute the dispersion distance between the generated and real-world sets. A smaller dispersion gap indicates that the generated code more authentically captures the syntactic and structural variance found in practice.

Table 5 reports the dispersion gaps for both tools relative to GitHub. S-MRVA variants consistently yield a markedly smaller gap (e.g., 0.2377 for CodeT5p) compared to Statfier (0.2974). This difference is strongly statistically significant ($p < 0.001$ across all models), underscoring S-MRVA's ability to reflect genuine real-world diversity. As a result, S-MRVA evaluates SAST tools against a realistically varied landscape of code patterns rather than relying on repetitive, superficial mutations.

6.4 RQ4: Generalization on Sparse Vulnerabilities

This RQ evaluates the generalization capability of S-MRVA across uncommon vulnerabilities, which are typically underrepresented in real-world codebases.

The *long-tail sparsity* of vulnerability patterns poses a fundamental challenge for mining-based variant generation. Figure 4

[12] https://github.com/semgrep/semgrep-rules/pull/3798
[13] https://github.com/semgrep/semgrep/issues/11483

Table 6: Test Coverage and Mutant Number Deviation Comparison vs. MRVA, on CodeQL Subset. "Deviation" indicates the standard deviation of the number of mutants per rule.

Metric	MRVA	S-MRVA
Test Cov.	37 (77.1%)	**46 (95.8%)**
Deviation	155.64	**7.02**

Table 7: Ablation Results (mean ± s.d. over 5 repeated runs). Mutants and defects are reported per run (/run) and normalized per 1,000 mutants (/1K). Red cells indicate statistically significant degradation vs. S-MRVA ($p < 0.01$).

Config	Mut.	Rep.		Uniq.		ER_{ud}(%)
		/run	/1K	/run	/1K	
S-MRVA-R	790.0 ±51.9	31.2 ±7.7	39.4 ±8.8	25.8 ±5.4	32.6 ±6.0	83.2 ±4.6
S-MRVA-S	580.0 ±36.7	32.4 ±7.8	55.9 ±13.1	19.6 ±3.0	33.9 ±5.6	61.7 ±7.9
S-MRVA	545.4 ±35.4	32.4 ±3.6	59.5 ±6.1	28.6 ±3.4	52.5 ±6.1	88.2 ±2.3

illustrates the significantly skewed distribution of MRVA's repository mining findings: the top 80% of findings originate from just 7 rules, while 11 rules yield zero findings. Crucially, these top 7 prevalent rules all possess low severity scores (< 8), whereas the sparse tail accounts for all high-severity rules (scores 8–10). This indicates that mining-based approaches like MRVA suffer from an inherent *severity bias*—generating abundant test variants for trivial flaws but leaving rare, high-severity vulnerabilities severely under-tested.

Conversely, S-MRVA's compositional synthesis approach is resilient to this long-tail distribution. By retrieving and assembling isolated sub-patterns rather than relying on holistic pattern occurrences, it generates variants uniformly across all rules. As detailed in Table 6, S-MRVA achieves a superior test coverage metric of 95.8% (46/48 rules) compared to MRVA's 77.1% (37/48). The two rules missed by S-MRVA resulted strictly from broken initial seeds provided by CodeQL, which triggered an early pipeline exit. Furthermore, S-MRVA is also more stable. The standard deviation in mutant generation across rules plummets from a massive 155.64 (MRVA) to 7.02 (S-MRVA), demonstrating that S-MRVA provides consistently dense and uniform test coverage, successfully mitigating the severity and frequency biases that plague mining approaches.

6.5 RQ5: Ablation Study

To isolate the contribution of each key component, we design two variants of S-MRVA, corresponding to the two core stages in our pipeline overview (see Figure 2). The first variant is *S-MRVA-R*, which disables the logic-tree refinement loop in the Semantic Decomposition stage (i.e., skip Step B ② in Figure 2, directly using the initial decomposition $\mathcal{L}_{init}$ as the final sub-pattern set), and *S-MRVA-S*, which replaces the Targeted Retrieval and Variant Synthesis stage (i.e., Step C in Figure 2) with an LLM-based mutation. Specifically, given the targeted sub-pattern, it prompts the LLM to generate functional-preserving variants directly from parametric knowledge, without retrieving real-world code.

We evaluate each variant on the same 10% of rules uniformly sampled from the five rule subsets as used in previous RQs. To account for non-determinism, we repeat each experiment five times independently. For each configuration, we report the mean and standard deviation across the five runs for each metric. We present the results from two complementary perspectives to facilitate interpretation of the repeated measurements. The *per-run* metrics (labeled **/run**) report the absolute count of defects produced in a single run, reflecting the pipeline's overall throughput. The *per-1,000-mutants* metrics (labeled **/1K**) normalize these counts by the total number of mutants generated in that run (*i.e.*, $/1K = \frac{/run}{Mut.} \times 1000$), measuring how many defects the pipeline obtains per thousand mutants. All outputs from the three configurations were pooled, anonymized, and manually reviewed following the same verification protocol as RQ1. We further performed a paired t-test across the results to assess statistical significance, comparing each ablation against the full pipeline. Table 7 reports all results.

First, S-MRVA-R generates 44% more mutants than the full pipeline (790 vs. 545) yet yields slightly fewer unique defects per run (25.8 vs. 28.6) and substantially fewer per 1,000 mutants (32.6 vs. 52.5, $p < 0.01$). The Rep./run metric (initial reports before verification) shows no significant difference, but Rep./1K does ($p < 0.01$), indicating that refinement improves reporting efficiency rather than expanding total reports. Manual inspection reveals that without refinement, the initial logic tree is fragmented into many fine-grained sub-patterns. Generating mutants independently for each such sub-pattern produces a large number of redundant variants that probe the same weaknesses repeatedly. In contrast, S-MRVA consolidates related sub-patterns, producing fewer duplicates and more distinct defect-revealing mutants. Second, S-MRVA-S maintains a similar mutant count (580 vs. 545) and a similar number of reported defects (Rep.), but drops significantly in unique defects (Uniq.) per run (19.6 vs. 28.6) and per 1K mutants (33.9 vs. 52.5), as well as ER_{ud} (61.7% vs. 88.2%). The reason is that Rep. counts all triggered discrepancies before manual verification, including the adversarial constructs that S-MRVA-S frequently produces (e.g., assembling wildcard addresses via `('.'.join(['0','0','0','0']), port)`). These unnatural reports are filtered out in Uniq. and ER_{ud}, revealing that removing our real-world code synthesis design will yield substantially fewer actionable, realistic defects.

In summary, S-MRVA-R sacrifices budget on duplicate predicates while S-MRVA-S sacrifices quality on unnatural mutants. Only the full pipeline preserves both high efficiency (52.5 unique defects per 1K mutants) and high fidelity (ER_{ud} 88.2%).

6.6 RQ6: Sensitivity of Parameters

S-MRVA involves two key hyperparameters: the refinement bound and the deduplication threshold τ. The refinement bound controls the scope of sub-pattern decomposition, directly influencing the efficiency and effectiveness of mutant generation. The threshold τ governs the clustering granularity of retrieved GitHub snippets, balancing candidate compression against defect recall. To systematically assess their impact and justify our default choices for the main experiments, we conduct sensitivity analyses on both parameters.

6.6.1 Refinement Bound. Our ablation study in Section 6.5 showed that the refinement improves efficiency compared to S-MRVA-R. To

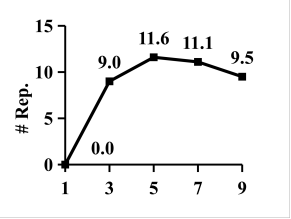


Figure 5: Sensitivity Analysis Result of Bound B_{high}.

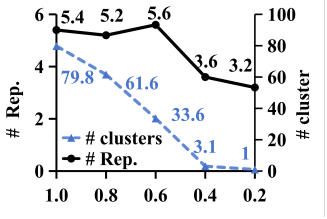


Figure 6: Sensitivity Analysis Results of threshold τ.

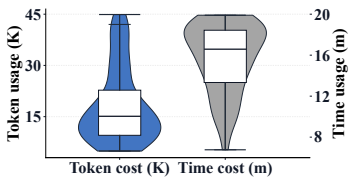


Figure 7: Cost of Tokens and Time to Generate a Variant.

Table 8: Comparison across Different LLM Backends.

Model	Pot.	Rep.	Uniq.
QF	99	45	16
MM	157	62	28
DS	130	61	39
QP	**183**	**59**	**46**

understand how the bound thresholds affect this efficiency gain, we analyze the sensitivity of B_{high}. In our design, $B_{\text{low}} = 2$ serves as a quality gate that ensures the decomposition yields at least two sub-patterns (Section 4.1). Therefore, we fix it and vary B_{high}={1, 3, 5, 7, 9} to study the cost versus effectiveness tradeoff. To mitigate non-determinism, we repeat each B_{high} setting five times. To keep the experimental cost manageable (5 B_{high} values × 5 runs, totaling 25 runs per rule), we select representative rules for this case study. Specifically, we compute the number of predicates in the initial logic tree for every rule in our benchmark, divide the rules by this count into four quartiles, and randomly sample one rule per quartile. This ensures that our experiment covers a broad spectrum of rule complexity in each repeat. For each B_{high} value, we report the mean number of reported defects per 100 mutants (# Rep.).

Figure 5 shows the evaluation results. With $B_{\text{high}} = 1$, the bound interval is unsatisfiable ($B_{\text{low}} = 2$), so the refinement loop terminates with no valid decomposition and yields no mutants. Raising the bound to 3 restores production (9.0). The efficiency peaks at $B_{\text{high}} = 5$ at 11.6. Further loosening the bound to 7 and 9 reduces efficiency to 11.1 and 9.5, respectively, suggesting that admitting more predicates introduces redundancy without increasing detected defects. Therefore, we adopt $B_{\text{high}} = 5$ by default, as it consistently achieves overall peak efficiency across all sampled rules.

6.6.2 Deduplication Threshold τ. As described in Section 4.2, S-MRVA clusters raw retrieved snippets from GitHub by embedding similarity and keeps one representative per cluster. The threshold τ controls the granularity of such compression. To study the cost versus effectiveness tradeoff of τ, we use the same subset and the same 5-repeat settings as the B_{high} sensitivity experiment. For each τ value, i.e., 1.0, 0.8, 0.6, 0.4, 0.2, we report the average number of retained candidates (# clusters) and the average number of reported defects (# Rep.). Figure 6 shows the evaluation results. With $\tau = 1.0$, all candidates are retained (79.8 per query), yielding 5.4 reported defects. Lowering τ to 0.8 removes about a quarter of the candidates (i.e., 61.6) with a marginal drop in reported defects (i.e., 5.2). Compressing further to 0.6 removes another 45% of the candidates (i.e., 33.6) with no drop in reported defects (i.e., 5.6), making $\tau = 0.6$ strictly more efficient than 0.8. Below 0.6, however, the reported defects drop to 3.6 at $\tau = 0.4$ and 3.2 at $\tau = 0.2$, indicating that overly aggressive merging eliminates entire defect patterns rather than just duplicates. We found that $\tau = 0.6$ consistently preserves the reported defects while removing more than half of the candidates compared with the $\tau = 1.0$ baseline. Therefore, for the main experiments in this paper, we adopt $\tau = 0.6$ as the default.

6.7 RQ7: Overhead and Extensibility

6.7.1 Token and Time Cost. Figure 7 presents the distribution of token costs and generation times per variant. S-MRVA leverages a map-reduce model to process each targeted SAST rule. After decomposing a rule into independent sub-patterns, the synthesis and verification of variants proceed in parallel. The preliminary analysis and decomposition of each SAST rule incur a one-time cost ranging from 20K to 60K tokens, depending on rule complexity and format. This cost is amortized across the total number of variants generated. On average, S-MRVA consumes approximately 17K tokens per variant. Regarding time overhead, with a provider throughput of 24 tokens/s and GitHub API rate limits of 6 req/m, the average generation time is approximately 16 minutes per variant.

6.7.2 Extensibility across SAST tools and LLMs. In S-MRVA, each SAST target is encapsulated as a modular tool within the agent framework. Adapting S-MRVA to a new analyzer (e.g., migrating from Semgrep to Fortify) requires implementing bindings for the command-line interface within an abstract analyzer class. Our implementation requires approximately 350 lines of Python code per SAST tool to handle code scanning and output parsing.

6.7.3 Impact of LLM Backend Choice. Table 8 compares four LLM backends on 20 randomly sampled rules. Qwen3.5-Plus (QP) achieves the highest unique defect result (46 defects, $ER_{ud} = 78.0\%$). Its budget sibling, Qwen3.5-Flash (QF) (50% lower cost), yields only 16 defects. DeepSeek-V3.2 (DS), at a comparable price, reports more initial findings but fewer unique defects (39). Minimax-M2.5 (MM), at roughly double the cost, delivers only 28 unique defects. We find that the performance gap stems primarily from accuracy during variant verification; weaker models more frequently misclassify variants. As a direction for future work, we plan to explore employing a dedicated security model or an agent-based approach as a more reliable judge in this verification step.

7 Threats to Validity

Internal Validity. A primary internal threat lies in the reliability of the LLM used for context synthesis and oracle generation. LLMs can hallucinate or generate semantically incorrect code. We mitigated this by explicitly constraining the LLM's task to small, minimized snippets and by designing a rule-specific judgment prompt. Furthermore, all unique defects were manually verified by the authors before submission to the tool maintainers.

External Validity. The external validity of our findings may be limited by the selection of target SAST tools and the specific rules

evaluated; the results may not generalize to all possible static analyzers or entirely novel vulnerability types. Additionally, the real-world snippets are sourced from GitHub; while vast, they may not perfectly represent proprietary, closed-source corporate codebases. Furthermore, the randomness of LLMs and updates to the code repository on GitHub can lead to the possibility of not generating completely identical variants.

Construct Validity. We measure realism through code provenance (real GitHub snippets), perplexity, and embedding-space dispersion against a GitHub baseline. Perplexity captures token-level predictability, which correlates with but does not independently guarantee realism; a syntactically repetitive snippet can score well without reflecting genuine engineering practice. We therefore triangulate through three signals: (i) provenance from real repositories, (ii) low perplexity as a necessary filter against synthetic constructs, and (iii) dispersion proximity to real GitHub code as a structural diversity check. The ultimate validator is maintainer-confirmed impact: our variants have intercepted new true vulnerabilities and reduced false alerts in production.

Reproducibility. The pipeline is non-deterministic due to the inherent stochasticity of LLMs. To control this and guarantee reproducibility, we intervene at multiple levels. First, we set temperature to zero to reduce output variance. At the workflow layer, S-MRVA uses LangGraph's stateful pipeline to combine LLM calls with deterministic algorithms. We also implement core operations (e.g., merging and splitting the logic tree) entirely in deterministic code rather than offloading them to the LLM. These design choices collectively ensure reproducible sub-pattern decomposition, stable synthesis, and consistent verification. We also conducted 5-repeat experiments in RQ5 and RQ6 for reproducible results.

8 Related Work

Static Analyzer Evaluation and Benchmarking. The evaluation of SAST tools has traditionally relied on curated benchmarks [2, 9, 21, 24, 31]. Widely used suites such as Juliet [2] and OWASP Benchmark [30] provide a baseline for comparing analyzer precision and recall. However, these synthetic suites often fail to capture the complexity of real-world codebases. Recent efforts have shifted toward mining CVEs and corresponding bug-fix commits from open-source repositories to create more realistic datasets, such as Vul4J [3]. While these benchmarks are valuable, they are static and do not provide the continuous, targeted feedback required to iteratively improve SAST rules. S-MRVA complements these datasets by dynamically generating realistic, rule-specific test programs from real-world code snippets, enabling continuous, fine-grained feedback on SAST development.

Automated Test Generation for Analyzers. Borrowing techniques from compiler testing [5–8, 16, 20, 39], researchers have proposed fuzzing approaches with differential testing or metamorphic testing oracles to automatically evaluate static analyzers. For instance, Statfier [42] and StaAgent [25] apply semantic-preserving transformations to existing test cases to uncover inconsistencies in SAST tool behavior. However, while effective at finding certain synthetic edge-case bugs, the approaches often generate unnatural code structures (e.g., trivially dead branches) that lack realism and cannot reflect vulnerabilities in the wild. In contrast, S-MRVA explicitly prioritizes realism by retrieving and synthesizing real-world code snippets, ensuring that the generated test cases represent realistic coding practices. Other previous work leverages fault injection to generate mutants [14, 15, 23, 26]. For example, SA_bugs injects certain vulnerabilities (e.g., out-of-memory, null-pointer-dereference) into Csmith-generated code to test C static analyzers. While these techniques can generate mutants at scale, they are limited to the predefined set of injectable bugs and struggle to capture the complex, context-dependent vulnerabilities (e.g., SQL injection, cross-site scripting) that modern SAST tools are designed to detect.

9 Conclusion

This paper addresses the fundamental bottleneck in SAST rule validation: existing methods cannot simultaneously achieve realism and scalability. We observe that while complete vulnerability instances are rare, their constituent semantic sub-patterns are abundant in real-world code. Leveraging this insight, we propose a divide-and-conquer paradigm that decomposes SAST rules, retrieves realistic code snippets for each sub-pattern, and synthesizes them into real-world grounded variants. We implement this approach in S-MRVA and evaluate it across 220 rules from four mainstream SAST tools. S-MRVA uncovered 458 unique flaws and achieved 97.27% defect coverage, far exceeding existing baselines. To date, we have reported 89 high-severity cases with a 95.2% confirmation rate. Industrial security teams have used our variants to intercept new true vulnerabilities and eliminate false alerts in production.

Acknowledgments

We thank all the anonymous reviewers for their insightful feedback. We also thank the CodeQL, Bandit, Semgrep, and SpotBugs communities for their positive feedback and support for our work. This work is supported by the National Natural Science Foundation of China (No.62622210 and No.U2436207).

References

[1] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2024. "False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing. In *Proceedings of the 45th IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 3979–3997. doi:10.1109/SP54263.2024.00019

[2] Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and Java Test Suite. *Computer* 45, 10 (Oct. 2012), 88–90. doi:10.1109/MC.2012.345

[3] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 464–468. doi:10.1145/3524842.3528482

[4] Checkmarx. 2026. Checkmarx Application Security Platform. https://checkmarx.com/. Accessed: 2026-04.

[5] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 700–711. doi:10.1109/ICSE.2017.70

[6] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (2021), 36 pages. doi:10.1145/3363562

[7] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543

[8] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proceedings of the 4th NASA Formal Methods Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012*. Springer, 120–125. doi:10.1007/978-3-642-28891-3_12

[9] Doaa Dalaq, Kaniz Fatima Daya, Alaa Dalaq, Muhammed Nazmul Arefin, and Mahmood Khan Niazi. 2025. A Systematic Literature Review on Static Application Security Testing (SAST) Tools: Evaluation, Benchmarks, Challenges, and Future Directions. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering Companion, EASE Companion 2025, Istanbul, Turkey, June 17-20, 2025*. ACM, 162–168. doi:10.1145/3727967.3756838

[10] GitHub. 2026. CodeQL. https://codeql.github.com/. Accessed: 2026-04.

[11] GitHub. 2026. GitHub Code Search. https://github.com/features/code-search. Accessed: 2026-04.

[12] GitHub Security Lab. 2026. MRVA: Multi-Repository Variant Analysis. https://github.com/GitHubSecurityLab/gh-mrva. Accessed: 2026-04.

[13] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Trans. Software Eng.* 49, 12 (2023), 5154–5188. doi:10.1109/TSE.2023.3329667

[14] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1656–1678. doi:10.1145/3660781

[15] Yu Hu, Zekun Shen, and Brendan Dolan-Gavitt. 2022. Characterizing and Improving Bug-Finders with Synthetic Bugs. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 971–982. doi:10.1109/SANER53432.2022.00115

[16] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 43–55. doi:10.1109/ICSE48619.2023.00016

[17] Zongze Jiang, Ming Wen, Ge Wen, and Hai Jin. 2025. Fact-Aligned and Template-Constrained Static Analyzer Rule Enhancement with LLMs. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, November 16-20, 2025*. IEEE, 1642–1654. doi:10.1109/ASE63991.2025.00138

[18] Rajeswari Hita Kambhamettu, John Billos, Tomi Oluwaseun-Apo, Benjamin Gafford, Rohan Padhye, and Vincent J. Hellendoorn. 2022. On the Naturalness of Fuzzer-Generated Code. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 506–510. doi:10.1145/3524842.3527972

[19] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 150–162. doi:10.1109/ICSE43902.2021.00026

[20] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 239–250. doi:10.1145/3293882.3330553

[21] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. 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, 921–933. doi:10.1145/3611643.3616262

[22] Yuan Li, Peisen Yao, Kan Yu, Chengpeng Wang, Yaoyang Ye, Song Li, Meng Luo, Yepang Liu, and Kui Ren. 2025. Understanding Industry Perspectives of Static Application Security Testing (SAST) Evaluation. *Proc. ACM Softw. Eng.* 2, FSE (2025), 3033–3056. doi:10.1145/3729404

[23] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 544–555. doi:10.1145/3533767.3534380

[24] Austin Mordahl. 2023. Automatic Testing and Benchmarking for Configurable 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, 1532–1536. doi:10.1145/3597926.3605232

[25] Elijah Nnorom, Md Basim Uddin Ahmed, Jiho Shin, Hung Viet Pham, and Song Wang. 2025. StaAgent: An Agentic Framework for Testing Static Analyzers. *CoRR* abs/2507.15892 (2025). arXiv:2507.15892 doi:10.48550/ARXIV.2507.15892

[26] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 1097–1109. doi:10.1145/3540250.3549128

[27] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. 2021. On the TOCTOU Problem in Remote Attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS 2021, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 2921–2936. doi:10.1145/3460120.3484532

[28] OpenText. 2026. OpenText Static Application Security Testing (Fortify). https://www.opentext.com/products/static-application-security-testing. Accessed: 2026-04.

[29] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 42 (Jan. 2025), 28 pages. doi:10.1145/3697010

[30] OWASP. 2026. OWASP Benchmark. https://github.com/OWASP-Benchmark. Accessed: 2026-04.

[31] Ivan Pashchenko, Stanislav Dashevskyi, and Fabio Massacci. 2017. Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*. IEEE Computer Society, 163–168. doi:10.1109/ESEM.2017.24

[32] Julian Aron Prenner and Romain Robbes. 2024. Out of Context: How important is Local Context in Neural Program Repair?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, Article 83, 13 pages. doi:10.1145/3597503.3639086

[33] PyCQA. 2026. Bandit: A Tool Designed to Find Common Security Issues in Python Code. https://github.com/PyCQA/bandit. Accessed: 2026-04.

[34] Semgrep. 2026. Semgrep App Security Platform. https://semgrep.dev/. Accessed: 2026-04.

[35] Mingjie Shen, Akul Abhilash Pillai, Brian A. Yuan, James C. Davis, and Aravind Machiry. 2025. Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper). *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 1077–1100. doi:10.1145/3728923

[36] SonarSource. 2026. SonarQube. https://www.sonarsource.com/products/sonarqube/. Accessed: 2026-04.

[37] David A. Tomassi. 2018. Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 4-9, 2018*. ACM, 980–982. doi:10.1145/3236024.3275439

[38] Stefan Wagner, Marvin Muñoz Barón, Davide Falessi, and Sebastian Baltes. 2025. Towards Evaluation Guidelines for Empirical Studies Involving LLMs. In *Proceedings of the 2nd IEEE/ACM International Workshop on Methodological Issues with Empirical Studies in Software Engineering, WSESE@ICSE 2025, Ottawa, ON, Canada, May 3, 2025*. IEEE, 24–27. doi:10.1109/WSESE66602.2025.00011

[39] Zifan Xie, Ming Wen, Shiyu Qiu, and Hai Jin. 2024. Validating JVM Compilers via Maximizing Optimization Interactions. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*. ACM, 345–360. doi:10.1145/3622781.3674188

[40] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 283–294. doi:10.1145/1993498.1993532

[41] Alperen Yildiz, Sin G. Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil Mon Divakaran. 2025. Benchmarking LLMs and LLM-based Agents in Practical Vulnerability Detection for Code Repositories. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Association for Computational Linguistics, 30848–30865. https://aclanthology.org/2025.acl-long.1490/

[42] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. 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, 237–249. doi:10.1145/3611643.3616272

[43] Huaien Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan. 2024. Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*. ACM, 1772–1784. doi:10.1145/3650212.3680398

[44] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 47–51. doi:10.1145/3639476.3639762

[45] Jingyun Zhu, Kaixuan Li, Sen Chen, Lingling Fan, Junjie Wang, and Xiaofei Xie. 2024. A Comprehensive Study on Static Application Security Testing (SAST) Tools for Android. *IEEE Trans. Software Eng.* 50, 12 (2024), 3385–3402. doi:10.1109/TSE.2024.3488041