Patch-Guided Vulnerability Detection: Extracting Java API Security Rules via Attack–Defense Cross-Analysis

Bofei Chen[1], *Shuang Liao*[1], *Lei Zhang*[*1], *Chibin Zhang*[2], *Mathias Payer*[2], *Yuan Zhang*[1]
[1] *Fudan University,* [2] *EPFL*

Abstract

Security-sensitive APIs are critical components in modern Java applications, yet improper usage of these APIs frequently leads to severe vulnerabilities such as remote code execution. Existing methods for generating API security rules are limited as they rely on incomplete documentation or infer patterns from source code based on discovered inconsistencies.

We introduce VULGENIE, a patch-driven framework that extracts precise API security rules from confirmed security patches to then detect API misuse vulnerabilities. VULGENIE addresses three key challenges. First, it isolates violated constraints and defenses-related changes from noisy patches using our novel modification behavior dependency patch graph datastructure. Second, it identifies protected security-sensitive APIs and synthesizes rules through attack-defense cross-validation. Third, it scales analysis with adaptive, deviation-guided static analysis to balance precision and performance. Evaluated on 150 recent Java security patches, VULGENIE extracts 198 correct API security rules with 81.82% precision, uncovering 177 rules absent in CodeQL. On ten popular Java applications, VULGENIE detects 46 0-day vulnerabilities, substantially outperforming state-of-the-art works. Through our responsible vulnerability disclosure, 26 vulnerabilities have already been fixed with ten CVE identifiers assigned.

1 Introduction

Empirical evidence [23] reveals a harsh reality: incorrect API usage causes 52% of Java bugs. Application Programming Interfaces (APIs) enable modularity and reusability but require precise adherence to usage rules to prevent (security) vulnerabilities. Among the vast ecosystem of Java APIs, certain security-sensitive APIs (SAPIs) pose particular risks. These APIs must be invoked under strict security constraints, collectively referred to as API security rules (ASR), which govern valid values and behaviors for the receiver, arguments, and return values. When developers violate these rules, they inadvertently create attack vectors that allow attackers to exploit SAPIs for unintended operations such as remote code execution (RCE), data exfiltration, or system compromise.

For example, Hutool [17], a popular Java utility library with 30k GitHub stars, contained a call to evaluate expressions without proper validation. This lacking security check enabled attackers to craft malicious expressions that execute arbitrary code on the target system, resulting in a critical remote code execution vulnerability affecting thousands of Maven projects (CVE-2023-24163 [33]). This case exemplifies how seemingly minor API misuse can have devastating security consequences, turning a legitimate functionality into a dangerous attack vector.

Given the prevalence and severity of API misuse vulnerabilities, researchers have developed various approaches to automatically extract API security rules and detect misuse. However, existing techniques face fundamental limitations that hinder their effectiveness in practice. Documentation-based approaches [18, 29, 34, 44, 45] extract usage constraints from API reference manuals and tutorials, but suffer from a critical flaw: API documentation typically emphasizes intended functionality while frequently omitting security constraints, leading to incomplete and outdated rules. Source code inconsistency analysis [20, 25, 40, 43] infers correct usage patterns from large codebases, assuming frequent usage reflects correct patterns. However, this majority-vote heuristic fails catastrophically when insecure usage dominates codebases, as often occurs before vulnerabilities are discovered and fixed. These fundamental limitations stem from a common weakness: existing approaches lack access to verified examples of what constitutes exploitable API misuse and how such misuse should be properly mitigated.

We propose a fundamentally different approach that leverages security patches as a rich, verified source of API security knowledge. Our key insight is that security patches serve as confirmed artifacts that implicitly encode precise fix patterns and security rules. Unlike documentation or heuristic analysis, patches represent real-world evidence of exploitable

*Lei Zhang is the corresponding author

```
01 public class SwitchTaskUtils {
02 +   private static final Set<String> blackKeySet = Sets.newHashSet("invoke", "eval", ...);
03      public static String generateContentWithTaskParams(String condition, ...) {
04          String content = condition.replaceAll("'", "\'"); ...
05 +        for (String blackKey : blackKeySet) {
06 +            if (content.contains(blackKey)) {
07 +                throw new IllegalArgumentException("not valid" + condition);}} ...
08          return content;}}                    Violated Constraints : Missing Input Sanitization
09 String content = SwitchTaskUtils.generateContentWithTaskParams(info.getCondition(), ...); ...
10 result = SwitchTaskUtils.evaluate(content); // Security-Sensitive APIs
```

Figure 1: Simplified code example of violated constraints in the patch for CVE-2024-23320.

vulnerabilities and their concrete remediation strategies, with code diffs providing empirical evidence of violated security constraints and the specific SAPIs that were misused. We define the violated security constraints as predicates over an SAPI's receiver, arguments, or return values that must hold to safely invoke the SAPI but are violated in the vulnerable (pre-patch) code. Such violations introduce the necessary exploit conditions. For example, Figure 1 shows how the patch for CVE-2024-23320 adds validation code (Line 2, 5–7) that exposes a previously violated security constraint: before fixed, user-controllable input `content` was passed to the SAPI `evaluate` (Line 10) without undergoing the required checks. As a result, the pre-patch program state violates the corresponding security constraint, thereby resulting in an expression injection vulnerability. The key code diffs enable us to precisely identify the violated constraint(s) and associate them with the corresponding misused SAPI(s).

Our patch-driven approach significantly improves precision over analyzing full source code: rather than inferring patterns from potentially noisy or inconsistent usage, we focus directly on confirmed security fixes that demonstrate both vulnerable and secure usage patterns. However, achieving this vision poses three fundamental technical challenges: First, violated constraints and defenses extraction requires isolating security-relevant changes from noisy patches that frequently contain extensive unrelated and multi-purpose modifications, which can mislead violated constraints and defenses identification. Second, SAPI discovery must identify protected security-sensitive APIs through indirect mitigation or violated constraints, as our empirical analysis shows that only 23.2% of patches directly modify SAPI calls. Instead, patches usually adjust surrounding code whose effects propagate to SAPIs through complex interprocedural flows. This makes straightforward control/data-flow analysis imprecise, often returning numerous unrelated candidate APIs. Third, scalable analysis in API misuse vulnerability detection requires a delicate balance between precision and efficiency. Detecting high-impact Java API misuses often hinges on precise usage-pattern auditing and source-to-sink tracing. While high-precision techniques (e.g., path- and context-sensitive analysis) are indispensable for accurate detection, applying them at scale can quickly become prohibitively expensive, leading to memory exhaustion or analysis timeouts.

In this paper, we present VulGenie, a novel framework that systematically addresses these challenges through an approach with three stages. Stage I tackles violations and defenses extraction by constructing a Modification Behavior Graph (MBG) that models dependencies among patch modifications, enabling intelligent isolation of security-relevant changes from noise and extraction of violated constraints and defenses. Stage II addresses SAPI discovery through attack-defense cross-validation that traces non-local dependencies *across methods and even libraries*, identifying SAPIs even when not directly modified in patches, and synthesizes precise API security rules using LLM assistance. Stage III achieves scalable analysis by integrating extracted rules into adaptive static analysis that employs attack-defense intersection and deviation-guided strategies to balance precision and performance, focusing computational resources on critical paths while maintaining detection accuracy. By bridging the gap between patch analysis and practical vulnerability detection, VulGenie demonstrates significant improvements in detecting API misuses across multiple methods and modules.

To evaluate the API security rule extraction capability of VulGenie, we construct a benchmark consisting of 150 recently disclosed Java security patches from popular open-source Java applications (detailed in §4.1). From these 150 patches, VulGenie successfully extracts correct 198 API security rules with a precision of 81.82%, as manually verified by security experts. Importantly, 177 of these rules are not covered by CodeQL's [1] existing knowledge, demonstrating VulGenie's capability to uncover previously undisclosed security rules. We further assess the effectiveness of these rules and VulGenie's vulnerability detection capability on ten popular Java applications in their latest versions. VulGenie detects 46 0-day vulnerabilities. We responsibly reported our findings to related developers and received their confirmation. So far, 26 vulnerabilities have been fixed, with ten CVE identifiers assigned. As a comparison, CodeQL and GraphiMuse [31] detect only 18 and three vulnerabilities, respectively.

We summarize the contributions of this paper as below:

- We propose a novel Java API security rules generation approach from complex, real-world security patches, enabling automatic integration of the latest API misuse rules into vulnerability detectors.
- We introduce a novel attack-defense intersection and deviation-guided path- and context-sensitive analysis to precisely and efficiently detect misuse vulnerabilities.
- VulGenie generated 177 previously unknown rules relative to CodeQL's existing knowledge base, based on 150 recent disclosed Java security patches.
- VulGenie discovered 46 confirmed 0-day vulnerabilities based on our extracted rules. To date, developers have fixed 26 vulnerabilities and assigned ten CVE IDs.

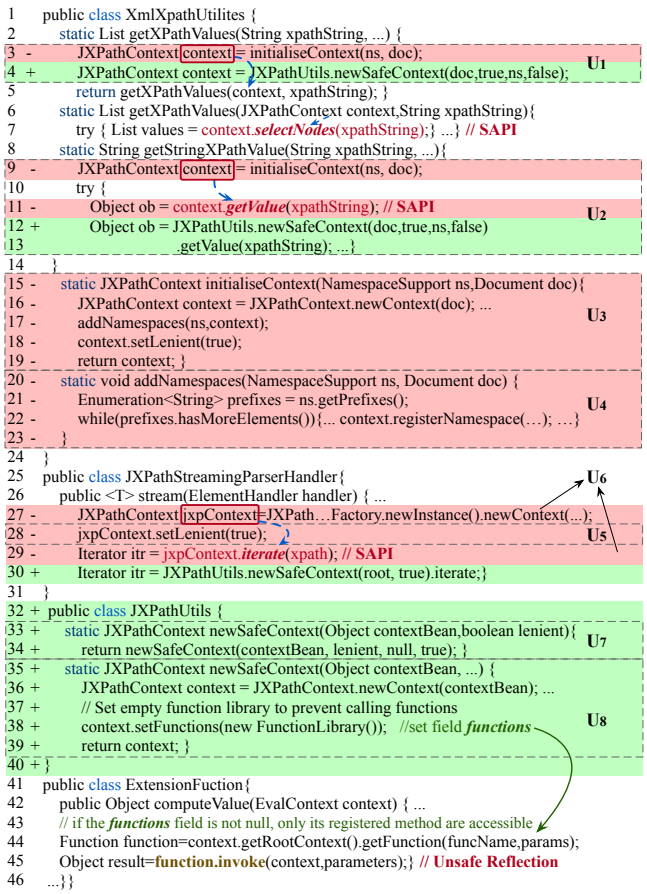


Figure 2: The simplified patch for CVE-2024-36404 (The original patch contains 263 lines of code modification.)

2 Overview

2.1 A Motivating Example

Our motivating example uses CVE-2024-36404, an expression injection vulnerability in the widely-used GeoTools [3] geospatial library. This case presents an interesting scenario where existing approaches fail. Neither JXPath's API documentation nor the GeoTools source code provides security guidance for the misused security-sensitive APIs (SAPIs): selectNodes (Line 7), getValue (Line 11), and iterate (Line 29) in Figure 1, plus additional APIs like selectSingleNode that share the same vulnerability but were not explicitly patched.

Our approach overcomes this limitation by deducing API security rules directly from the security patch. This methodology enables us to identify the root cause of the vulnerability in JXPath's SAPIs and extract generalized security rules applicable beyond the immediate patch context. Using these rules, we then discovered two zero-day vulnerabilities in Convertigo [15], a downstream application, which have been assigned one CVE ID after we reported them to the developers.

API Security Rule Example. Since `selectNodes`, `getValue`, and `iterate` are patched semantically equivalent in Figure 2, we generate similar security rules for all three. We use `selectNodes` as a representative example. Figure 3 presents the extracted security rule, which consists of three parts: ❶ SAPI signature: identifies `selectNodes`, which parses user-supplied expressions and can dynamically invoke embedded methods, exposing an attack surface. ❷ Exploit condition: the method call becomes exploitable when: (a) the first argument (the expression string) is user-controllable; and (b) the evaluation context method resolution without an explicit allowlist. ❸ Reference defense: disable method resolution by setting the context's functions field to a whitelist (Line 38 in the patch). Omitting this step results in insecure use of `selectNodes`, thereby violating the intended security policy. Next, after extracting these rules, we utilize them to detect new vulnerabilities in other projects. As an example, we successfully identified insecure invocations of `selectNodes` and `selectSingleNode` in Convertigo.

```
<ASR language="java">
    <VulnCategory>Expression Injection</VulnCategory>
    <VulSummary>…</VulSummary> // Vulnerability cause summary
    <SAPI> // Security-sensitive API (SAPI signature)
        <FunctionIdentifier>
            <NamespaceName>org.apache.commons.jxpath</NamespaceName>
            <ClassName>JXPathContext</ClassName>
            <FunctionName>selectNodes</FunctionName>
            <Parameters>
                <ParamType>java.lang.String</ParamType>
            </Parameters>
        </FunctionIdentifier>
        <Specifications> // Exploit Condition
            <Conditional>
                <ArgMark>1</ArgMark> // security-sensitive argument/
receiver index
                <ConditionalType>1,5</ConditionalType>
                <VulComment>An attacker-controlled xpath argument could include
malicious function calls, which would execute when this function is called. The old context allowed
function execution via the default function library.</VulComment>
                <SecurityComment>The patch configures JXPathContext with an
empty FunctionLibrary, preventing any function calls during xpath evaluation</
SecurityComment>
            </Conditional>
            <Conditional>
                <ArgMark>-1</ArgMark>
                <ConditionalType>2</ConditionalType>
                <VulComment>…</VulComment>
                <SecurityComment>…</SecurityComment>
            </Conditional>
        </Specifications>
        <Protections> // Reference Defense Strategy
            <Protection>
                <ArgMark>-1</ArgMark>
                <Comparator>None</Comparator>
                <InvokedFuncName>setFunctions</InvokedFuncName>
                <CompareValue>new FunctionLibrary()</CompareValue>
            </Protection>
        </Protections>
    </SAPI>
</ASR>
```

Figure 3: An example of SAPI (`selectNodes`) security rule.

Our work. We propose a patch-driven pipeline with three stages that converts security patches $P_{sec}s$ into rules of security-sensitive API usage.

❶ **Identify violated constraints & defenses**: Given a P_{sec}, we compute a semantic diff to identify the violated security constraints and the newly introduced repair actions. Upon identifying the corresponding SAPI, we concretize them into

the (i) exploit condition and the (ii) reference defense. For example, in Figure 2, the changes in Lines 3–4 and 38 reveal the violated constraint (no limits on reflectively invoked methods) and its concrete fix (calling `setFunctions` to set an empty whitelist in the `context`'s `functions` field). ❷ **Link constraints to security-sensitive APIs**: We then trace data and control flows from the violated constraints back to the relevant security-sensitive API (SAPI) calls, even when those calls are not directly modified by the patch. Continuing the example, data flow analysis from the *context* modified at Line 3 (in the pre-patched code) traces back to the SAPI `selectNodes`, linking the new allowlist requirement to that API. ❸ **Detect real-world vulnerabilities**: We next treat identified SAPIs as sinks and perform taint analysis on target programs to detect API misuse vulnerabilities, i.e., cases where insecure API usage coexists with exploitable source-to-sink paths that without/lack of reference defenses.

However, this workflow raises three key challenges. We discuss the challenges and introduce our key ideas (in § 2.2).

2.2 Challenges and Solution

Challenge#I: Violations and defenses extraction. Security patches often mix security fixes with unrelated changes, and security logic is fragmented across multiple methods. This dispersion creates noise that obscures the actual vulnerability fix, leading to incorrect SAPI identification. Existing approaches (e.g., APHP [24]) rely on heuristics syntactic patterns like conditional statements or method calls to identify violated constraints, making them prone to noise-induced errors. As shown in Figure 2, while the patch modifies 263 lines, merely 15 lines (e.g., Lines 3-4) add security constraints on the source-to-sink paths, with just one critical addition (Line 38) implementing the core defense. In contrast, unrelated modifications (e.g., Lines 15–23, 28) may misleadingly highlight well-protected APIs, e.g., `setLenient` or `registerNamespace`, causing downstream false positives.

Moreover, a single security patch often concurrently fixes multiple SAPI usages through interleaved code modifications. We must carefully separate them as interleaved changes for these interfaces can obscure each other. On one hand, it may cause certain vulnerable SAPIs to be overlooked during analysis, resulting false negatives. On the other hand, it can lead to the generation of overly broad security rules that incorrectly incorporate constraints meant for other SAPIs, ultimately reducing these rules' effectiveness.

Our Solution: Modification behavior dependency guided violated constraints and defenses extraction. From a preliminary study, we observe that seemingly scattered code modifications in a security patch are united by control/data-flow and structural dependencies that collectively expose the patch's corrective intent. First, code changes referencing the same SAPI tend to form cohesive yet minimally overlapping clusters, even when distributed across different methods or modules. For example, in Figure 2, VULGENIE partitions modifications into clusters by analyzing both dependency relations (e.g., U_1 invokes methods defined in U_3 and U_8) and semantic equivalence of modification behaviors (e.g., U_1, U_2, and U_6 all replace the original `JXPathContext` constructor wrapper `initializeContext` with the safer alternative `newSafeContext`). Based on this analysis, the patch is decomposed into three distinct clusters: $\{U_2, U_{3\text{-}4}, U_8\}$, $\{U_6, U_{3\text{-}4}, U_{7\text{-}8}\}$, and $\{U_1, U_{3\text{-}4}, U_8\}$, each corresponding to a specific SAPI fix. In addition, within each cluster, the dependency topology differentiates functional roles: units frequently referenced by others typically serve as centralized, reusable defenses, whereas those referencing many others are more directly tied to violated security constraints. For example, VULGENIE it selects U_8 (Line 38 in Figure 2), referenced by $U_{1,2,6}$, as the core defense, and flags U_1 (Lines 3–4) as the call site that activates this defense, thereby exposing the violated constraint.

Challenge #II: SAPI discovery. SAPIs are often not directly modified in patches but are linked to fixes through interprocedural flows. Our study shows only 23.2% of SAPIs appear directly in patch diffs.

On one hand, the violated constraints may have control- or data-flow connections to many method calls in the diff code context, which could potentially be SAPIs. However, the absence of general validation techniques often results considerable false positives. For example, APHP filters methods by extracting method names mentioned in the patch description. However, such information is usually unavailable in practice. On the other hand, a patch may fail to cover all *APIs* that share the same vulnerability root cause (i.e., Line 45 in Figure 2, a user-controllable insecure reflection), leaving homologous vulnerabilities partially unaddressed. For instance, if the SAPI `selectSingleNode` is not extracted, a vulnerability in Convertigo that exploits it could go undetected.

Our Solution: Attack-Defense Cross-Validated SAPI Inference. Our key insight is that the value-flow associated with violated constraints in the pre-patched version reveal source-to-sink paths along which insecure data propagates. By analyzing where these insecure paths intersect with the defense logic introduced in the patch, we can accurately identify the program interfaces (SAPIs) that were fixed and gain a deeper understanding of the underlying vulnerability.

For example, the selected patch unit U_1 exposes an exploit condition in which the variable `context` lacks associated security constraints. Thus, it performs a backward and forward data- and control-flow analysis to trace `context` from its initialization (at Line 3 in Figure 2) to related method calls, e.g., `addNamespaces` (Line 17), `setLenient` (Line 18), `selectNodes` (at Line 7). Meanwhile, VULGENIE leverages LLMs to identify defense logic from U_8, i.e., the allowlist configuration embedded in the `functions` field of the `context`. By analyzing the intersections between attack-related and defense-related value flows, i.e., along the path

from `selectNodes` to `getFunction` at Line 44 in Figure 2), VULGENIE can successfully eliminate two false positives, e.g., `addNamespaces`, `setLenient`. In addition, by performing a backward analysis from `computeValue`, it further identifies SAPIs that share the same root cause but were not explicitly modified in the patch, such as `selectSingleNode`.

Challenge #III: Scalable analysis. Unlike most existing API misuse detection works (e.g., [24, 31]) that primarily focus on local API misuse patterns, we target practically exploitable vulnerabilities by capturing data-flow paths from user-controlled variables pass to insecure SAPI invocations in real-world programs. To improve detection accuracy, it is essential to apply path- and context-sensitive analysis, which enables more precise tracking of data flows and allows distinguishing paths that are already protected by security mechanisms from those that are unprotected or potentially bypassed.

However, program paths can grow exponentially, leading to the well-known path explosion problem. In addition, Java's advanced features (e.g., polymorphism) often obscure vulnerabilities within complex data- and control-flows, generating long call chains. Together, these factors make precise path- and context-sensitive analysis computationally expensive and prone to timeouts or memory exhaustion. As demonstrated in §4.2, our evaluation indicates that the false negative rate due to out-of-memory exceptions and timeouts reached 84.78%. To manage the trade-off between analysis precision and efficiency, prior works [11,30,35] typically adopt selective strategies, applying path-/context-sensitive analysis only to method callsites/paths relevant to analysis targets. These approaches often predefine rules (e.g., k-limit [11], code patterns [30,35]); however, we found that these rules tend to be scenario-specific. Balancing analytical precision with overhead in Java misuse vulnerability detection remains a significant challenge.

Our Solution: Intersection and Deviation-guided Path- and Context-sensitive Analysis for Misuse Detection. We introduce a novel attack-defense intersection and deviation guided strategy to conduct selective path- and context-sensitive taint analysis. The strategy originates from two key observations. (a) Intersection-guided path-sensitive analysis. We observe that security-relevant flows often diverge at intersections, where one branch enforces a defense while the complementary branches can bypass it and reach a sink. These mutually exclusive conditions form unmergeable control-flow focal points that are critical for precise reasoning. By concentrating path-sensitive analysis on these paths, VULGENIE explicitly separates defended from undefended flows, pruning false positives on protected paths and false negatives on bypassed ones. (b) Deviation-guided context-sensitive analysis. Another key observation is that call sites along source-to–sink/defense paths often dominate on detection precision. Accordingly, VULGENIE applies full context-sensitivity for the analyzed method on these critical paths and adaptively reduces sensitivity for methods that deviate, improving precision on security-critical flows without compromising scalability.

3 Design

In this section, we describe the details of our approach, called VULGENIE. Figure 4 presents the architecture of VULGENIE, which consists of three core modules. The input of VULGENIE is a security patch (i.e., P_{sec}). VULGENIE first construct a MBG to capture dependencies of modified code in P_{sec}, and uses it to isolate each SAPI remediation, preciously point the violated constraint(s) and corresponding defense(s) (§ 3.1). Later, VULGENIE trace both data and control flows from violated constraints and defense to the corresponding SAPI calls, and employs a LLM-assisted method to summary exploit conditions and reference defenses, and ultimately synthesizes the API security rules (§ 3.2). Afterward, VULGENIE treats SAPIs as sinks and applies adaptive, path- and context-sensitive taint analysis from predefined sources to detect Java vulnerabilities that satisfy the exploit conditions and are not mitigated by reference defenses (§ 3.3).

3.1 Violations and Defenses Extraction.

In stage I, given a P_{sec}, VULGENIE first constructs the Modification Behavior Graph (MBG) to capture both consistent remediation intent and the distinct roles of fine-grained patch units (i.e., a piece of diff code). Guided by the MBG, VULGENIE then isolates each SAPI-specific fix and extracts the violated constraints along with the corresponding defenses (i.e., a piece of code to fix the insecure usage of the SAPI).

We conduct a study of 100 security patches affecting popular applications and find that seemingly scattered code modifications within a patch are typically connected through control-flow, data-flow, and structural dependencies, revealing the patch's corrective intent. Building on this observation, the study also finds that repair units targeting the same SAPI usage are semantically inequivalent (179 of 187 SAPI-fixes), while units targeting different SAPIs but sharing the same vulnerability root cause frequently exhibit semantic equivalence (36 of 37 patches). For example, in Figure 2, the repair units U_1 for `getValue` (Line 9) and U_2 for `selectNodes` (Line 7) share equivalent semantics, while the units U_1 and U_8 that both repair `getValue` differ semantically.

These findings motivate the design. We can construct a MBG to capture both the dependencies and semantic-equivalence relationships of repair units. The MBG is built on top of the program dependency graph (PDG), and we customize its node representation and edge semantics to encode patch-level information that standard PDGs do not capture. MBG nodes represent fine-grained diff-code segments rather than individual statements, and MBG edges model their data-flow, control-flow, and structural dependencies, along with semantic-equivalence relationships. We then use the semantic-equivalence edges in the MBG to slice P_{sec} into SAPI-specific fixes (i.e., SMBGs). Next, we could identify the violated constraints and their corresponding defenses based on the depen-

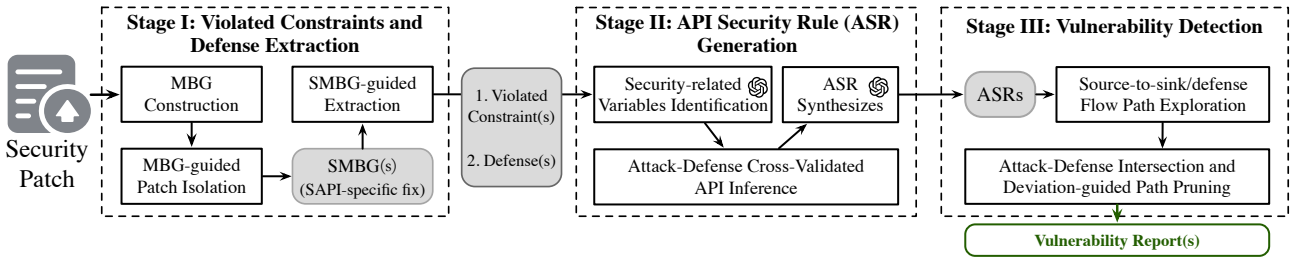


Figure 4: The Overall Architecture of VULGENIE with Three Stages.

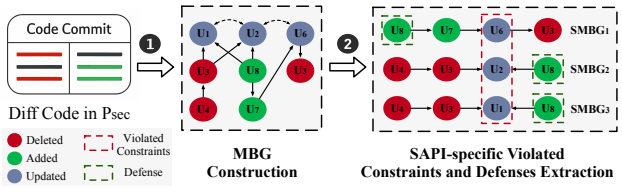


Figure 5: MBG construction and violated constraints and defenses extraction.

dencies within each SMBG.

Below, we first adopt graph-theoretic notation to formalize the MBG, and then detail how to construct such MBG.

Definition 1 (MBG Node). Let $\mathcal{U}$ denote the set of MBG nodes, where each node represents a minimal diff segment encapsulating a single repair action. For example, $U_{1,2,6}$ in Figure 2 capture the replacement of the constructor wrapper `initialiseContext` with a safer `newSafeContext`, whereas $U_{7,8}$ represent two added `newSafeContext` method declarations with different parameters.

Definition 2 (MBG Edge). The MBG edge set $\mathcal{E}$ contains two types of edges: dependency and semantic-equivalence edges.

❶ **Dependency edges** capture three types of relationships among code elements within patch units: 1) Data-flow edge (u_1, u_2) indicate that a variable in u_1 is propagated to u_2 via assignment, pointer dereference, or interprocedural flows through actual/formal parameters, receivers and return values. 2) Control-flow edge (u_1, u_2) means that the execution of statement or method invocation in u_2 depends on the outcome or declaration in u_1. 3) Structural relationship edge (u_1, u_2) indicate that the class/method in u_2 inherits or overrides the one in u_1. These edges allow us to group repair units that collectively implement a cohesive fix.

❷ **Semantic-equivalence edges** link patch units (MBG nodes) that exhibit equivalent repair behavior, enabling the separation of units that are connected by dependency edges yet address distinct SAPI repairs.

Then, we detail how VULGENIE constructs the MBG.

MBG node extraction. VULGENIE first decomposes the modified statements in P_{sec} into semantically atomic repair units (i.e., $\mathcal{U}$). These units isolate logically independent changes, preserving their intended repair semantics while avoiding inter-unit interference. Specifically, it employs a generalized repair pattern to partition P_{sec}, ensuring applicability across diverse vulnerability types. We identify ten representative repair strategies that capture different high-level semantics of how the diff code mitigates vulnerabilities, as summarized in Table 5. This classification is consistent with prior studies [39, 42] and also account for Java specific language features. For patch in Figure 2, VULGENIE first uses a diff tool [2] to extract modified code hunks (e.g., Lines 3–4, 9, 11–13), identifies high-level repair patterns in each block, and then partitions P_{sec} into semantically atomic patch units by pairing deleted and added statements that share both the same pattern and modification target. For example, the deleted hunk at Lines 15–23 is split into two independent "change method declaration" units (i.e., U_3, U_4), while the modified hunk containing both deleted and added code at Lines 3–4 is treated as a single unit (i.e., U_1). Finally, it prunes non-security-related edits (e.g., logging) based on code-structure heuristics, following patterns summarized by DISPATCH [36].

MBG edge establishment. From the extracted $\mathcal{U}$, VULGENIE first performs a static taint analysis to construct *dependency edges* among units. It then computes the pairwise similarity between all units and augments the unit graph with *semantic-equivalence edges*. To more precisely evaluate the semantic similarity of modifications between patch units and mitigate the influence of namespace or syntactic differences across repair blocks, we capture the patch pattern and modified elements' properties within each patch unit rather than raw text. Specifically, each unit is summarized into a signature consisting of two sets of hashed elements: (i) H_{pat}. Repair pattern (defined in Table 5); (ii) $\sum H_{chg}(u_i)$. the change trajectories, where each trajectory records the modified property and the hash values of its before- and after-states. VULGENIE applies specialized rules depending on the code element: (a) Variable: track changes to type, value or constructor (wrapper), and modifiers; (b) Method call: track changes in method signatures, parameter or receiver types, and return values; and (c) class: track changes to modifiers, superclasses, and implemented interfaces. Using U_1 in Figure 2 as an example, VULGENIE detects that the constructor wrapper of `context` changes from `initialiseContext` to `newSafeContext`, and encodes this as a trajectory. Parameter changes are recorded in the same manner. Based on these signatures, it computes the pairwise similarity using the weighted Jaccard index [4] of their hash sets, and classifies U_1, U_2,

and U_6 as semantically equivalent, as their similarity exceeds threshold (0.5 empirically).

VulGenie then constructs the MBG based on the identified $\mathcal{U}$ and $\mathcal{E}$. For example, Figure 5 illustrates the MBG of patch as shown in Figure 2.

MBG-guided SAPI-specific Patch Isolation. Since a single security patch can repair multiple SAPIs concurrently through interleaved code changes, careful separation of these modifications is essential: without isolating them, interleaved updates can obscure one another, causing vulnerable interfaces to be overlooked and producing overly broad rules that conflate constraints across APIs. To address this challenge, after constructing the MBG, VulGenie uses it to partition P_{sec} into SAPI-specific subgraphs (SMBGs) based on Algorithm 2 (in Appendix). Each SMBG is formed by grouping units connected via *dependency edges* while cutting along *semantic-equivalence edges*, which mark mutually exclusive implementations. Starting from a $u \in \mathcal{U}$ with minimal out-degree, VulGenie performs a depth-first expansion along dependency edges, adding reachable vertices only if no semantically equivalent node is already in the subgraph; the procedure then repeats with new roots until all units are assigned. The resulting maximal connected components are complete yet minimal repair fragments for individual SAPIs. For example, P_{sec} in Figure 2 can be decomposed into three SMBGs: beginning at U_1 and and excluding its semantically equivalent peers U_2, U_6 yields U_1, U_3, U_4, U_8 (SMBG3 in Figure 5); starting at U_2 yields $\{U_2, U_3, U_4, U_8\}$ (SMBG2); starting at U_6 yields $\{U_6, U_7, U_8, U_3\}$ (SMBG1).

Sub MBG (SMBG)-guided violations and defenses identification. First, VulGenie applies a PageRank-based algorithm to prioritize units that expose violated constraints and corresponding defenses, leveraging the inter-node dependency topology within each SMBG. PageRank is well-suited because a SMBG is a directed graph in which edges, analogous to hyperlinks, represent information flow. In PageRank, a node's score reflects its importance based on citations from other high-scoring nodes. In our setting, units with high out-degree typically encapsulate centralized defense logic reused across multiple paths, while units with high in-degree often lie near SAPI call sites orchestrating those defenses to address violated constraints. After scoring all units in each SMBG, VulGenie prioritizes them in descending and ascending order to identify those most relevant to violated constraints and defenses, respectively. As shown in Figure 5, VulGenie selects U_1 (Lines 3–4 in Figure 2) as revealing the violated constraint, and U_8 (Lines 35–39 in Figure 2) as exposing the defense logic within SMBG3.

This module effectively correlates complex, fragmented modification logic, enabling precise extraction of violated constraints and defenses. As demonstrated in §4.4, the raw patches had an average of 224.0 modified lines. After MBG-guided isolation, 126.8 security-irrelevant lines were removed, and SMBG-guided identification accurately extracted an average of 17.9 lines of violated constraints and 15.2 lines of defenses for each insecure SAPI usage. Without this, precision and recall decrease by 48.12% and 77.47%, respectively.

3.2 API Security Rule Generation

In Stage II, VulGenie first identifies attack- and defense-related variables from the violated constraints and defenses, respectively. It then performs value-flow tracing from these variables across the pre- and post-patched versions to uncover source-to-sink and source-to-defense paths that interrupt the reachability of the attack flow. By applying an advanced *attack-defense value-flow cross-validation technique*, VulGenie infers the relevant SAPIs. Finally, it leverages an LLM-assisted approach to summarize exploit conditions, reference defenses, and synthesize API Security Rules (ASRs).

❶ Core Task For LLMs

```
Analyze the provided security patch code and:
1. Identify the core security mechanisms that directly block vulnerability exploitation
2. For each mechanism, extract:
  - The security-related variables and methods implementing the defense
  - Estimate the defense level of security-related variables or functions (rate from 0 to 10)
  - The defense types of security-related variables or functions, including : (1) sanity checks; (2)
string sanitization; (3) object property settings (e.g., security attributes); (4) variable boundary
constraints; (5) global security variables. You can add other defense types as needed.
3. Prioritize mechanisms that neutralize code execution risks.
```

❷ Code From U_8 of SMBG3

```
[Commit]: a diff of Java code changes. Lines prefixed with - indicate removed (old, vulnerable)
code, and lines prefixed with + indicate added (new, fixed) code.
```java
+   @SuppressWarnings("unchecked")
+   public static JXPathContext newSafeContext(
+         Object contextBean, boolean lenient, NamespaceSupport ns, boolean declared) {
+     JXPathContext context = JXPathContext.newContext(contextBean);
+     context.setLenient(lenient);
+     if (ns != null) {
+       Enumeration<String> prefixes = declared ? ns.getDeclaredPrefixes() : ns.getPrefixes();
+       while (prefixes.hasMoreElements()) {
+         String prefix = prefixes.nextElement();
+         String uri = ns.getURI(prefix);
+         context.registerNamespace(prefix, uri); }}
+     // Set empty function library to prevent calling functions
+     context.setFunctions(new FunctionLibrary());
+     return context;}}
```
```

❸ Output Requirements

```
Format the output as XML with this structure:
```xml
  <DefenseMechanism>
    <SecurityImpact></SecurityImpact> # Explain how to block attack
    <Variables>
      <Variable>
        <isLocal></isLocal> # true: local variable; false: objects' field
        <Name></Name> # Defense Variable Name
        <BelongMethodName></BelongMethodName> # The Method Name
        <RelatedMethodName></RelatedMethodName> # Related Method Name
        <Level></Level> # Defense Level
        <Type></Type> # Defense Type
      </Variable> ... # Other Defense Variables
    </Variables>
  </DefenseMechanism>
```
Return only the data in XML format, no other words.
```

Figure 6: Prompt used to extract defense-related variables from U_8 (in Figure 2) of SMBG3 (in Figure 5).

Security-Related Variables Identification. VulGenie first identifies security-sensitive variables from both violated constraints (attack-related) and defense logic (defense-related). To avoid conflating them, the identification is field-sensitive, distinguishing different fields of the same object (e.g.,

the attack-related variable `context` vs. the defense-related `context.functions`). This enables precise separation of attack and defense value flows.

(a) Attack-related variables identification. VULGENIE conservatively extracts attack-related variables from violated constraints to minimize false negatives. It initially considers all variables within violations as attack-related, then filters them through cross-validation. To improve efficiency, it maintains an empirical knowledge base that records attack-related variables validated through final checks, along with corresponding vulnerability types. In later analyses of similar vulnerabilities, it prioritizes variables with matching method names and types as attack-related, streamlining the process.

(b) Defense-related variables identification. VULGENIE leverages LLMs and SMBG-guided code slicing to identify defense-related variables, reducing false positives in later analyses. Our key insight is that LLMs can generalize common coding patterns, capturing defense logic that traditional static analyses often miss due to syntactic variations. Leveraging the code property graph, VULGENIE performs bidirectional slicing (forward and backward) on both pre- and post-patched versions to obtain a precise slice of the defense code context, retaining only statements that are data-flow dependent on variables introduced or removed by the defense. Code within the same SMBG is prioritized to preserve semantically coherent context. The resulting slices are then provided to the LLMs (Figure 6), which extract three categories of defense elements: (1) local defense variables, (2) field defense variables, and (3) method calls that manipulate such variables. For the third category, VULGENIE employs static analysis to resolve the field variables manipulated by the identified methods, and marks them as defense-related variables. Taking U_8 (in Figure 2) as an example, Deepseek identifies `setFunctions`, and VULGENIE traces its effects to the `functions` field, which is then classified as a defense-related variable.

3.2.1 Attack-Defense Cross-Validated SAPI Inference

Next, VULGENIE performs taint analysis starting from the identified attack-related variables, treating their associated method invocations as candidate SAPIs. Simultaneously, it checks whether the attack value-flow is interrupted by the defense value-flow, removing candidate SAPIs along flows with no intersections to reduce false positives. To support this process, VULGENIE introduces a defense backward lifting algorithm for efficiently identifying the scope of patch-introduced defenses and a novel on-demand call graph completion-based cross-library analysis for deep tracing SAPIs across libraries.

Defense Backward Lifting. First, VULGENIE aims to identify the scope of patch-introduced defenses. However, initial defense-related variables often propagate across multiple methods or are embedded in upstream routines (e.g., sanitizers), making naive value-flow inference complex and prone to path explosion. To address this, VULGENIE performs backward analysis to lift these variables to upper layers of the call chain, thereby avoiding redundant traversal of complex call graphs. Algorithm 1 formalizes this process. For each defense-related variable identified, VULGENIE executes the following steps: (1) Field propagation. If the variable is a field, it locates all usage sites across methods within its class and marks those variables as new defenses-related targets (Lines 16–21). For example, analyzing `this.functions` from `setFunctions` triggers analysis of its usage in `getFunctions`. If the variable is assigned to a field, that field becomes a new defenses-related variable (Lines 12–14). (2) Parameter propagation. If the variable is passed as a method parameter, the corresponding arguments at the call sites are lifted (Lines 31–34). (3) Exception exit propagation. We observe that, in security patches, developers often mitigate vulnerabilities by adding exception handling or sanitizing sensitive values before return—both being termination-related behaviors. Accordingly, variables in the callers of such methods are treated as defense-relevant, as they are influenced by these protections (Lines 36–39). Note that identifying new defense-related variables through propagation does not in itself constitute a complete value-flow; rather, it incrementally expands the flow by discovering additional nodes, which are later linked together into a defense value-flow converging on an upstream anchor. If no new variables are identified through above steps, the current defense variable is marked as a top-level defenses-related variable for subsequent cross-validation (Lines 24, 42, 45). For example, in Figure 7, VULGENIE identifies a defense value-flow that originates from the field variable `functions` in `setFunctions` (Line 26) and converges on the `this` reference (the top-level, Line 7 in Figure 7) in `selectNodes`.

Figure 7: The attack-defense cross-validation process with two steps in SAPIs identification for CVE-2024-36404.

Algorithm 1 Defense Backward Lifting Algorithm

```
1: function IDENTIFYDEFENSEVARS(initVars)        ▷ Input: Initial defense-related
   variables initVars; Output: Top-level defense-related variables defenseVars
2:     defenseVars ← ∅
3:     stack ← initVars
4:     visited ← ∅
5:     while stack ≠ ∅ do
6:         var ← stack.pop()
7:         if var ∈ visited then
8:             continue
9:         end if
10:        visited ← visited ∪ {var}
11:        if ISFIELDRELATED(var) then
12:            assignedField ← GETASSIGNEDFIELD(var)
13:            if assignedField ≠ null then
14:                stack.push(assignedField)
15:            else
16:                belongClass ← GETBELONGCLASS(var)
17:                usagePoints ← GETUSAGE(belongClass, var)
18:                if usagePoints ≠ null then
19:                    for point ∈ usagePoints do
20:                        newVar ← GETFIELDVAR(point)
21:                        stack.push(newVar)
22:                    end for
23:                else
24:                    defenseVars ← defenseVars ∪ {var}
25:                end if
26:            end if
27:        else
28:            func ← GETBELONGFUNCTION(var)
29:            callSites ← FINDCALLSITES(func)
30:            if callSites ≠ null then
31:                if ISPARAMRELATED(var) then
32:                    for site ∈ callSites do
33:                        arg ← MAPTOACTUALARG(site, var)
34:                        stack.push(arg)
35:                    end for
36:                else if ISEXITRELATED(var) then
37:                    for site ∈ callSites do
38:                        receiver ← GETCALLERRECEIVER(site)
39:                        stack.push(receiver)
40:                    end for
41:                else
42:                    defenseVars ← defenseVars ∪ {var}
43:                end if
44:            else
45:                defenseVars ← defenseVars ∪ {var}
46:            end if
47:        end if
48:    end while
49:    return defenseVars
50: end function
```

Cross-library analysis. Another challenge stems from deeply nested invocations across multiple third-party libraries, which complicates the identification of attack-defense value-flow intersections due to control/data flow disruption caused by missing source code of the dependent third-part libraries.

To address this, VULGENIE adopts an on-demand call-graph refinement-based strategy. Instead of constructing call graphs for all dependent libraries upfront, VULGENIE starts with a lightweight call graph that contains only application code. When the analysis encounters a call site whose target resides in an external library, VULGENIE retrieves the corresponding JAR based on the method signature and dependency version from Maven, and incrementally resolves the missing call edges. Through this demand-driven refinement, VULGENIE gradually reconstructs complete cross-library call chains that enable precise tracing of method invocations across the application and its third-party libraries. This design enables VULGENIE to accurately recover attack- and defense-relevant call paths while improving analysis efficiency compared with whole-library call-graph construction.

We use the patch in Figure 7 to illustrate the inference process. First, VULGENIE identifies the defense-related variable `this` (Line 7) in `selectNodes`, which lies along the call chain without local call sites in JXPath. Next, it initiates forward taint analysis from the attack-related variable `context` (Line 2) in Geotools, treating associated method invocations as candidate SAPIs. As the analysis reaches the `selectNodes` call (Line 6), VULGENIE resolves its origin in JXPath by matching function signatures and consulting dependency information from the configuration file (e.g., `pom.xml`), dynamically downloading and integrating the required library as needed. This step reveals a value-flow connecting `context` and the `this` reference in `selectNodes`, confirming `context` (Line 5) as the attack-defense intersection. Finally, it collects the complete call chains spanning multiple libraries, from attack-related to defense-related variables, which serve as the methods for generating subsequent ASRs.

After confirming attack-defense intersections, VULGENIE locates the methods containing each candidate SAPIs' call sites and analyzes contextual information (e.g., argument types) to determine whether these caller methods should also be classified as SAPIs. This process allows for the iterative expansion of the SAPI set. For example, when analyzing the `computeValue` method (Line 17 in Figure 7), its caller `selectSingleNode` is identified as SAPI, ensuring that the final API security rules are both precise and comprehensive.

3.2.2 LLM-guided ASR Synthesizes

For each inferred SAPI, VULGENIE employs an LLM-assisted method to concretize the identified violated constrains and defenses (§ 3.1) into explicit exploit conditions and reference defenses, thereby synthesizing an API security rule (ASR). As illustrated in Figure 3, each ASR comprises three core components: ❶ a SAPI signature, which enables precise identification of the SAPI invocations (i.e., sinks) across different projects; ❷ exploit condition(s) and ❸ reference defense(s), which jointly characterize secure versus insecure usage patterns. By capturing such API-level rules that reflect the root causes of misuse rather than project-specific vulnerable-code patterns, VULGENIE can achieve strong cross-project generalization. This enables the detection of previously unknown vulnerabilities instead of merely rediscovering variants of known ones.

Below, we first describe the information encoded in each ASR component and how it contributes to detecting new vulnerabilities. We then introduce how to synthesize a complete ASR with the assistance of the LLM.

In an ASR, each exploit condition specifies the security-sensitive value positions (receiver, arguments, and return value) and their corresponding exploit condition types (as cat-

egorized in Table 1). In addition, it includes LLM-generated summaries (i.e., `VulComment` and `SecurityComment` in Figure 3) that abstract the underlying exploit root causes and defense strategies. These elements jointly indicate which argument(s) of the SAPI invocation VulGenie must inspect and in what manner. If a SAPI invocation (sink) along the source-to-sink path satisfies exploit conditions but lacks adequate defenses, VulGenie report as a vulnerability (detailed in §3.3). It evaluates existing defenses through a two-tiered process (detailed in § 3.3.1): exact matching against reference defenses extracted from patches (e.g., method names for security checks), and semantic matching against LLM summaries to recognize equivalent security checks that may be renamed or re-implemented across projects. To avoid false negatives, a path is classified as secure only when the LLM provides high-confidence evidence of sufficient defenses. As evaluated in § 4.4, this two-tiered approach filtered 39 false positives with no false negatives, leaving only three false positives, demonstrating cross-project generalizability and precision.

For each identified SAPI, VulGenie synthesizes an ASR by jointly leveraging static analysis and LLMs through two sub-steps. (i) It first identifies the security-relevant value positions in the SAPI invocation; and (ii) extracts relevant code context via static analysis to prompt a LLM to infer the corresponding exploit condition types and generate summaries of exploit root causes and defense strategies, as well as reference defenses. Specifically, VulGenie first identify the sensitive value positions through static taint analysis. Using Figure 7 as an example, it analyzes the data-flow of the attack-related variable `context` (Line 2) and determines that the sensitive value is the receiver (index -1) of the SAPI call (`selectNodes` at Line 5). Since a SAPI may involve additional security-relevant variables beyond those exposed in patch diffs, VulGenie then leverages an LLM to identify and supplement such missing variables. For example, when processing the `selectNodes` method (Line 5), while `context` (Line 2 in Figure 7, diff code at Line 3–4 in Figure 2) is automatically assigned as the security-relevant variable, VulGenie invokes the LLM to evaluate whether other variables, such as `xpathString` at Line 5 in Figure 7 (the first argument, index 0), also introduce the vulnerability. After expanding the variables, it generates the corresponding exploit conditions for all identified security-relevant variables.

Next, to ensure that the LLM focuses precisely on the target SAPI and improves the accuracy of its inferred condition types, summaries, and reference defenses, VulGenie employs static slicing to isolate: (1) the SAPI's method source code and (2) pre- and post-patch SAPI usage examples that highlight secure and insecure usage patterns. This slices away irrelevant code, constraining the LLM's reasoning to the code that truly reflects the SAPI's security semantics. For SAPIs located far from the patch diffs, meaningful pre- and post-patch usage examples are unavailable. To address this challenge, VulGenie begins rule synthesis from the SAPI closest to the diff code and then traverses along the call chain that intersects the attack- and defense-value flows. For each subsequent SAPI on the chain, VulGenie provides the LLM with (3) the caller's synthesized ASR and (4) the corresponding interprocedural source code between the two methods, guiding the LLM to infer how exploit conditions and defenses evolve along data- and control-flow transitions. For example, in Figure 7, VulGenie first synthesizes the ASR for `selectNodes`. This ASR then compensates for the lack of insecure and secure usage example pair when synthesizing the ASR for `iteratePointers` (Line 7 in Figure 7).

Table 1: Types of Exploit Condition.

Type	Condition Type	Description and Requirements
1	Miss/Insufficient Input Sanitization	Insufficient or missing checks (e.g., missing validation or sanitization)
2	Missing/Improper Permission Check	Missing/incorrect permission checks on security-sensitive variables, enabling unauthorized use within that scope
3	Improper Lifecycle Handling	Incorrect handling of the variable's lifecycle; lifecycle-related assertions must be verified
4	System-level Misconfiguration	Global security settings are incorrectly configured, e.g., disabled security policies
5	Just Controllable	Exploitability depends solely on whether the variable is attacker-controllable

3.3 Vulnerability Detection

In Stage III, VulGenie integrates extracted API security rules into a static analysis framework. Given a target program, it treats ASR-specified SAPI calls as sinks and conducts source-to-sink analysis to detect vulnerabilities. Each finding contains (a) the source-to-sink call chain and (b) value-flow(s) trace demonstrating bypass of defenses (e.g., sanitizers, if any), to facilitate subsequent manual review. There are two substeps: (i) lightweight exploration of source-to-sink flow paths, and (ii) fine-grained pruning of unreachable or already protected paths through an advanced selective, path- and context-sensitive taint analysis, which manages a balance between high precision and efficiency.

3.3.1 Path Exploration

Given a target program P, VulGenie first identifies candidate sinks by matching methods against *SAPIs' signature* defined in ASRs (synthesized in §3.2.2). It then labels existing defenses by checking if code statements contain the three key elements from ASR reference defenses, namely `comparator`, `invokedFuncName`, and `CompareValue` (e.g., Figure 3). VulGenie categorizes the identified defenses into two classes. The first class includes sanitization or control-flow constraints (Type 1, 3, 5 in Table 1) over user-controllable inputs. In the exploration step, VulGenie marks potential defenses along source-to-sink paths, and later, during fine-grained path pruning (in §3.3.2), verifies their effectiveness (i.e., whether they constrain the correct variables and are valid for the current data-flow path). The second class includes security configurations for SAPIs. Here, it filters out secure

usages from candidate SAPI calls: for system-level configurations (Type 4), it marks all related SAPI calls as secure, whereas for object-level configurations (Type 2), only those invoked with the specified configuration are secure.

Then, it performs a coarse-grained taint analysis to quickly enumerate all potential *source-to-sink flow path (SSFP)*, prioritizing recall over precision. Each path consists of a source-to-sink call path augmented with the source-related, path-insensitive data-flow information, and is annotated with the corresponding defenses (i.e., $\mathcal{D}$). For example, at T_7 (Line 23 in Figure 8), it merges the taints from different paths (Lines 16–18, 19–20, and 21–22) by computing the intersection of the taint sets. It thus records a source-to-sink flow path as `(channelInput, flag)` $\rightarrow ... \rightarrow T_3 \rightarrow ((C_{d1} \rightarrow T_4 \rightarrow D_1)/(C_1 \rightarrow T_5)/(C_2 \rightarrow T_6)) \rightarrow T_7 \rightarrow ... \rightarrow T_{11}(sink)$. It also records the source-to-defense flow path $\ldots \rightarrow T_9 \rightarrow D_2$.

Upon SSFP extraction, VULGENIE employs a linear-time, sign-guided method to compute *control-flow constraints* along two types of data-flow paths: (i) source-to-sink paths (denoted as C_es), and (ii) paths leading to existing defenses (denoted as C_ds). These constraints guide the rapid pruning of control-flow–unreachable paths and the selection of non-mergeable branch paths (§3.2.2). Specifically, for each conditional branch, VULGENIE assigns a unique identifier. A positive sign (H_p) indicates the path that executes when the condition is satisfied, whereas a negative sign ($-H_p$) indicates the path taken when it is not. Each node maintains the set of hash values of all predecessor path conditions. By combining the positive and negative sets, it efficiently derives the dominator conditions that must hold along source-to-sink paths or paths leading to defenses. This allows the system to quickly identify the path conditions that must be satisfied to invoke the next method along a source-to-sink path.

Based on $C_e s$, VULGENIE rapidly prunes control-flow unreachable call/data-flow paths. It first checks for contradictions among $C_e s$ and prunes call paths dominated by contradictory conditions. It then prunes data-flow paths that do not satisfy the $C_e s$. For example, in Figure 8, if C_{e_1} is not satisfied, the program takes the branch at Line 11, preventing reaching the sink at Line 29. Thus, it prunes the flow $... \rightarrow !C_{e1} \rightarrow T_1$. For C_ds, VULGENIE identifies the variables that determine the satisfaction of the conditions and marks the associated defense-relevant statements. For example, for C_{d2}, it identifies `checkFlag`, whose value must satisfy the condition required to invoke the sanitization at D_2. It then analyzes the assignment sites of `checkFlag` and marks the statements that can satisfy C_{d2} as defense-relevant operations, e.g., D_1 at Line 18.

3.3.2 Path Pruning

Then, VULGENIE applies a novel attack–defense intersection and deviation-guided, path- and context-sensitive taint analysis to prune unreachable or protected security-sensitive data-flows in each SSFP. If all flows are pruned, no report

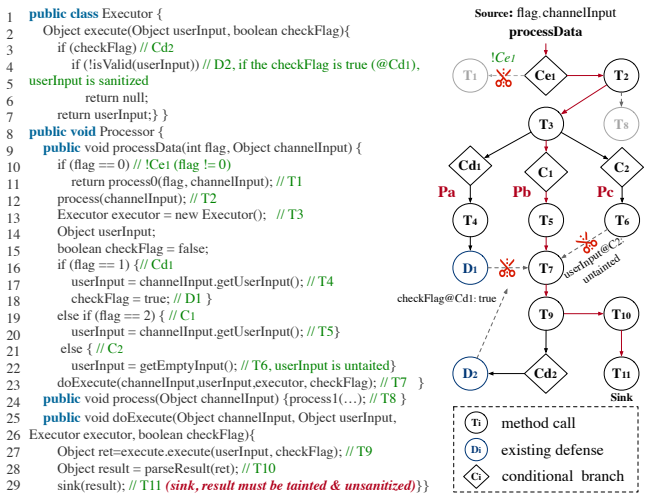


Figure 8: An example of Attack-Defense Intersection and Deviation-guided Path- and Context-Sensitive Analysis.

is generated; otherwise, VULGENIE reports the corresponding source-to-sink path with all exploitable flow paths. This analysis proceeds in two substeps: (i) it selects key paths that can distinguish flows blocked by existing defenses from those that may bypass them, using path-sensitive analysis; (ii) it re-executes a selectively path- and adaptively context-sensitive taint analysis along the SSFP to confirm unprotected control- and data-flow reachable paths.

Intersection-guided Path-Sensitive Analysis. VULGENIE retains path-sensitive only for paths that satisfies $C_d s$ to invoke existing defenses and their mutually exclusive execution paths, ensuring that it can distinguish paths that protected, missing or bypass the existing defenses. For example, C_{d_1} (Line 16 in Figure 8) is a must-satisfy (i.e., dominator) condition for invoking the defense. It maintains path-distinct abstract states along $C_{d1} \rightarrow T_4 \rightarrow D_1(P_a)$ and the alternative paths $C_1 \rightarrow T_5(P_b)$ and $C_2 \rightarrow T_6(P_c)$, without prematurely merging taint summaries at Line 23. As a result, the analysis preserves path-specific correlations between data taints and control predicates at Line 23. In particular, the abstract states of `userInput` and `checkFlag` are distinguished as follows: along P_a, `userInput` is tainted, while `checkFlag` is assigned true, which can satisfy the sanitization guard C_{d2} and thereby trigger input sanitization at Lines 4–6; along P_b, `userInput` remains tainted but `checkFlag` is false; along P_c, `userInput` is derived from a non-user-controlled source and is thus untainted. This selective preservation exposes defensive gaps while avoiding the computational cost of exhaustive path enumeration.

Adaptive Context-Sensitive Static Analysis. Context sensitivity is critical for precise static program analysis. In object-oriented programs (e.g., Java), object sensitivity has proven particularly effective for pointer and taint analyses [22,27,32], as it distinguishes method invocations based on their receiver objects, effectively treating the same method as multiple

context-specific instances. However, it is insufficient for two important cases: static method invocations and instance methods whose security-relevant behaviors depend on arguments rather than the receiver. A straightforward extension is to incorporate all arguments into the context abstraction. However, this strategy would dramatically increases the number of method summaries and propagated states, leading to context explosion and poor scalability.

To address this limitation, we extend object sensitivity with selective argument sensitivity to improve precision while controlling cost. Our key observation is that vulnerability exploitability is primarily influenced by how attacker-controllable variables propagate along source-to-sink paths and how defense checks gate these propagations. Accordingly, VULGENIE first leverages the detected source-to-sink flow paths (in §3.3.1) to identify, at a call site, those receivers and arguments whose values depend on security-sensitive sources or control relevant defenses, and incorporates only these security-relevant positions into the context abstraction. Call contexts are distinguished using the security-relevant provenance of the variables that reach these positions, so that only provenance elements affecting vulnerability exploitability or the execution of defenses contribute to context distinctions. In other words, VULGENIE distinguishes contexts only when security-relevant receivers or arguments reach the same call site through different security-relevant histories. For example, at the call site `doExecute` (Line 23 in Figure 8), the source-to-sink flow `channelInput` $\rightarrow ((T_4 \rightarrow D_1)/T_5)$ (`userInput = channelInput.getUserInput()`) $\rightarrow T_7$ marks `userInput` as a security-relevant argument. In addition, the source-to-defense flow through `isValid(userInput)` in `Executor.execute`, which is executed only when `checkFlag == true`, marks `checkFlag` as security-relevant because it controls whether the user input is validated before reaching the sink.

The next challenge is determining how much provenance to retain in order to distinguish different security-relevant calling contexts without incurring excessive computational cost. We represent a sensitive value-flow chain as a sequence of security-relevant program points (e.g., assignments from user inputs, calls to input sanitization, and updates to security-critical variables) along a taint-propagation path from user-controllable input(s) or defense-related variable(s) to a security-relevant receiver or argument at a call site. Such a chain records the security-relevant history of how the variable has been propagated and potentially sanitized; the more of this provenance we encode into the context, the more precisely we can distinguish different call contexts.

To control the trade-off between precision and cost, VULGENIE introduces a sensitivity level k, retaining only the last $\min(k,m)$ security-sensitive program points on a chain of length m as the provenance summary. Lower sensitivity may merge distinct calling contexts, introducing false positives or negatives, while higher sensitivity improves precision at the cost of increased overhead. This formulation naturally sets the stage for our deviation-guided strategy, which dynamically adjusts k to concentrate computational resources on the paths most critical for capturing exploitable behaviors.

Deviation-guided Strategy. To achieve high analysis precision while maintaining efficiency, VULGENIE adaptively adjusts the context-sensitivity level (k) based on deviations of the currently analyzed method across source-to-sink paths, defense paths, and defense-conflicting paths (i.e., *dominator paths*), prioritizing computational resources for analyses most critical to vulnerability exploitation and defense mechanisms.

Specifically, VULGENIE assigns the maximum context-sensitivity level K_{max} to each call site along dominator paths. Here, K_{max} is set to the length of longest sensitive value-flow chain for the selected security-relevant receiver or arguments on the corresponding source-to-sink flow. For example, along the flow `(channelInput,flag)` $\rightarrow T_2 \rightarrow ... \rightarrow T_{11}(sink)$ in Figure 8, the longest chain has length 5 (i.e., P_a: `(channelInput,flag)` $\rightarrow$ `userInput (= channelInput.getUserInput())` $\rightarrow$ `checkFlag=true` ($\wedge$ `isValid(userInput))` $\rightarrow$ `ret` $\rightarrow$ `result`), hence $K_{\max} = 5$. For any method m outside the dominator paths, it computes its shortest deviation to the nearest dominator paths, denoted as $dev(m)$, and assigns a reduced context-sensitivity level using $k = \text{round}(\alpha \cdot \sqrt{\frac{K_{max}}{dev(m)}})$, where α controls the impact of the deviation (default 1).

Taking Figure 8 as an example, when the execution reaches the sink, the sensitive value-flow chains on P_a, P_b and P_c differ in the provenance of `result`. Concretely, the chain on P_b is `(channelInput,flag)` $\rightarrow$ `userInput (= channelInput.getUserInput())` $\rightarrow$ `ret` $\rightarrow$ `result`, while the chain on P_c is `userInput (= getEmptyInput())` $\rightarrow$ `ret` $\rightarrow$ `result`, and the chain on P_a have been described above. Therefore, setting $k = 5$ retains all security-relevant program points on the chain suffixes, which precisely separates the three paths and allows VULGENIE to identify the sole exploitable path P_b while suppressing spurious paths such as P_a and P_c. In contrast, under a fixed small level such as $k = 2$, the provenance summaries of P_a, P_b and P_c collapse to the same suffix `[ret, result]`. As a result, VULGENIE can no longer distinguish the different origins (e.g., `userInput`) of `ret` on these three paths, which may introduce false positives or false negatives. On the other hand, always setting k to $K_{\max}$ across the program can incur substantial overhead by over-refining contexts even for methods far away from dominator paths. Our deviation-guided strategy avoids both extremes by allocating high sensitivity only to dominator-path call sites and progressively lowering k as the deviation to these paths increases. For example, the method call at T_8, which exhibits a deviation of 1, results in a reduction of its context sensitivity from K_m to 2.

4 Evaluation

4.1 Experimental Setup

Implementation. We implemented VULGENIE in Python and Java. Given that security patch diffs operate at the source code level, we developed the rule extraction module using Tree-sitter [37] and Joern [8]. Through Tree-sitter [37] we extract MBG nodes and implement source-code taint analysis using Joern's Code Property Graph (CPG) for edge construction and inter-procedural SAPI inference. Our customized PageRank implementation leverages NetworkX [7] to identify violations and defenses. The vulnerability detection module is based on JDD [13], a state-of-the-art frameworks tailored for static analysis and optimization of Java. We extend JDD's static analysis to identify source-to-sink flow paths and adaptive path- and context-sensitive taint analysis. We integrated DeepSeek [16] into VULGENIE in our experiments.

Table 2: Breakdown of our evaluation dataset for RQ1.

Open-source Applications	Version	# Stars	# LoCs
Apache Seata	2.1.0	25.7K	311,328
Apache Iotdb	2.0.2-1	5.8K	1,320,408
azkaban	3.81.0	4.5K	142,839
Undertow	2.3.14	3.7K	299,698
Snail-job	1.6.0	3.3K	75,548
JFinal	5.2.1	3.2K	60,250
Solon	2.7.3	2.6K	143,011
zt-zip	1.17	1.4K	12,924
RuoYi-Vue-fast	3.9.0	1.1K	32,553
Convertigo	8.3.5	0.4K	361,372

Experiments and Effectiveness Summary. To evaluate the effectiveness of VULGENIE, we conduct two experiments. First, we assess VULGENIE's capability to derive API security rules. To construct the dataset, we initially collected 930 security patches from popular open-source Java projects (i.e., GitHub/Gitee repositories with ≥ 500 stars or widely used Maven Central libraries [5] with ≥ 500 reverse dependencies, and then selected 150 patches for evaluation, restricting to CWE IDs (20,22,79,89,400,502,611,917) and requiring manual verifiability. From these, VULGENIE successfully generated 198 ASRs, 177 of which were previously unknown.

Next, we evaluate the effectiveness of the generated API security rules in detecting real-world API misuse vulnerabilities. Specifically, we randomly collect another active ten popular open-source Java projects from Github/Gitee with ≥ 500 stars and with $\geq 10,000$ lines (listed in Table 2) as analysis targets. VULGENIE successfully identified 46 zero-day vulnerabilities from them, all confirmed by corresponding vendors.

For easy understanding, we first present the vulnerabilities discovered by VULGENIE in §4.2 and §4.3, followed by its capability to extract API security rules in §4.4. Finally, we discuss the performance of VULGENIE in §4.5.

4.2 RQ1: Discovering Zero-day Vulnerabilities

In this section, we evaluate the effectiveness of VULGENIE in detecting API misuse vulnerabilities in Java applications, using the extracted API security rules detailed in §4.4.

Confirmed and Fixed Vulnerabilities Table 3 summarizes the overall results of the experiment. VULGENIE reported 55 potential vulnerabilities, among which nine were false positives, and each report includes two informative components: (i) violated security rules of SAPI, (ii) an source-to-sink path for attackers to exploit the SAPI. Through manual inspection of each case, we ultimately confirmed 46 taint style vulnerabilities, including Java deserialization (9), DoS (9), Message leak (2), Path travel (11), Expression injection (5), XXE (8), JNDI injection (1), and XSS (1). We followed responsible disclosure practices and reported these vulnerabilities to the relevant vendors. Up to now, all these vulnerabilities have been confirmed by developers and 26 of them have already been patched with ten CVE IDs assigned. By exploiting these vulnerabilities, attackers could launch serious attacks including arbitrary code execution, exfiltrate sensitive data, or upload and deploy malicious files to the target server. Thus, many of them are rated as high-severity: five of ten CVEs score 9.8.

False Positives. We identified two root causes for the nine false positives. First, VULGENIE focuses on identifying SAPIs and their security rules but uses pre-defined rules to identify sources during detection. Six FPs occurred because our predefined rules failed to accurately identify untrusted user input across diverse frameworks. For example, in Apache Seata, it identified a source in the client-side code that could not actually be exploited to attack the server. Second, three FPs were due to silently patched fixes whose defensive logic was not covered by our collected defense patterns.

Ablation Study#1. We conducted an ablation study to isolate the contributions of the attack-defense interaction and the deviation-guided strategy. Specifically, we compared VULGENIE against two variants: (i) VULGENIE-NoAPCS, which disables the selective strategy and instead enforces a fully context- and path-sensitive analysis; and (ii) VULGENIE-NoAP-1obj, which performs a path-insensitive yet k-object context-sensitive analysis, with k fixed to the widely adopted setting of 1, as suggested in prior work [22]. The results highlight the necessity of the attack-defense interaction and deviation-guided strategy in VULGENIE. VULGENIE-NoAPCS either timed out (30 minutes) or ran out of memory on seven projects, detecting only seven vulnerabilities. Its recall dropped to merely 15.22%, despite a seemingly high precision of 87.50%. In contrast, VULGENIE-NoAP-1obj successfully completed all analyses and achieved full recall (detecting all 46 vulnerabilities). However, its 1-object context-sensitive strategy proved inadequate for the complex and lengthy call chains in Java applications, resulting in 159 false positives and a precision of only 22.44%.

Table 3: Comparison among GraphiMuse, CodeQL, and VULGENIE (our approach) with detailed metrics per application.

Application	Vulnerabilities	CodeQL‡					GraphiMuse					VulGenie‡				
		TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
Apache Seata	3	0	26	3	0.00%	0.00%	0	439	3	0.00%	0.00%	3	1	0	75.00%	100.00%
Apache Iotdb†	10	5	9	5	35.71%	45.45%	-	-	-	-	-	10	0	0	100.00%	100.00%
azkaban†	2	0	3	2	0.00%	0.00%	-	-	-	-	-	2	0	0	100.00%	100.00%
Solon	5	3	12	2	20.00%	60.00%	1	346	4	0.29%	20.00%	5	1	0	83.33%	100.00%
Undertow	2	0	3	2	0.00%	0.00%	0	463	2	0.00%	0.00%	2	2	0	50.00%	100.00%
jFinal	2	0	0	2	0.00%	0.00%	0	57	2	0.00%	0.00%	2	2	0	50.00%	100.00%
zt-zip	4	4	1	0	80.00%	100.00%	1	3	3	25.00%	25.00%	4	0	0	100.00%	100.00%
Rouyi-Vue-fast	1	0	22	1	0.00%	0.00%	0	2	1	0.00%	0.00%	1	0	0	100.00%	100.00%
Snail-job	9	0	0	9	0.00%	0.00%	0	257	7	0.00%	0.00%	9	3	0	75.00%	100.00%
Convertigo	8	6	38	2	12.00%	75.00%	1	1689	1	0.06%	50.00%	8	0	0	100.00%	100.00%
Total	46	18	114	28	13.64%	39.13%	3	3,256	43	0.09%	6.50%	46	9	0	83.64%	100.00%

‡ We treat each unique source–sink pair as an independent vulnerability.
† GraphiMuse ran out of memory on these components and produced no analysis results..

4.3 RQ2: Compared with state-of-the art tools

We compare VULGENIE with two state-of-the-art tools: (i) CodeQL, a widely adopted static analysis tool for large-scale vulnerability detection; and (ii) GraphiMuse, the most related work leveraging code-consistency analysis to detect Java API misuse bugs. Prior works such as APHP [24] and RTFM [29], which target API misuse detection in C/C++ via patches or documentation, are excluded as inapplicable to Java.

Since labeling all vulnerabilities is infeasible, we construct ground truth from vulnerabilities detected by all three tools [38]. Each vulnerability was validated with a proof-of-concept, yielding 46 verified cases.

Result Overview. Table 3 provides a detailed comparison of the effectiveness of VULGENIE, CodeQL and GraphiMuse. Overall, VULGENIE clearly outperforms both baselines, achieving 70.00% higher precision and 60.87% higher recall than CodeQL, and 81.73% higher precision and 93.50% higher recall than GraphiMuse. Against a ground truth of 46 vulnerabilities, VULGENIE detected all of them, whereas CodeQL identified only 18 and produced 114 false positives.

False Positive and Negative Analysis. After analyzing CodeQL and GraphiMuse's false positives and negatives, we examined the root causes behind VULGENIE 's superior performance. First, compared with CodeQL, VULGENIE can automatically identify recently disclosed SAPIs and distill more fine-grained defense patterns from patches. CodeQL integrates a set of classic SAPI security rules by default, but these manually maintained rules lag behind newly disclosed components and the latest exploitation techniques, leading to 28 false negatives. Furthermore, CodeQL lacks support for developer-customized defenses, leading to various false positives. For example, in RuoYi-Vue-Fast, file-upload and download operations have implemented directory-traversal checks, but CodeQL still reports 22 related false positives since it cannot recognize sanitizers. Second, relative to GraphiMuse, VULGENIE offers two key advantages. First, GraphiMuse assumes that correct usage patterns of SAPIs occur more frequently than incorrect ones. This heuristic incorrectly flaged 3,457 correct usages whose business contexts produce inconsistent patterns and missed 43 SAPI calls whose correct usage is infrequent. VULGENIE avoids this pitfall by deriving precise rules from code changes in P_{sec}s that contain verified API usage violations. Second, GraphiMuse lacks taint analysis from user- controllable inputs to SAPI calls, producing eight additional false positives that VULGENIE avoids.

To illustrate the benefits of our patch-based SAPI rule update mechanism for Java API misuse detection, we present a 0-day vulnerability discovered by VULGENIE for case study.

Case Study: QLExpression Injection Vulnerability. VULGENIE uncovered a critical expression injection vulnerability enabling remote code execution (RCE) in Snail Job (Gitee 3.4K stars), a widely used distributed retry and scheduling framework. VULGENIE detected this vulnerability via a security rule we abstracted from a security patch [9] in QLExpress [9]. The rule codifies the required preconditions for safe use of the SAPI `QLExpression.execute`. Specifically, when the expression (first argument) is user-controlled, the QLExpress engine must be initialized with sandboxing and constructor restrictions, e.g., enable the sandbox via QLExpressRunStrategy.setForbidInvokeSecurityRiskConstructors(true) and configure constructor allow/deny lists with QLExpressRunStrategy.addRiskSecureConstructor or QLExpressRunStrategy.addSecureConstructor. Snail Job violates this API security contract by evaluating user-controllable expressions without these defenses, thus VULGENIE reported it as a vulnerability. CodeQL failed to detect this vulnerability due to the lack of the API rule, and GraphMuse missed it because `ExpressRunner.execute` is invoked only once in Snail-Job.

4.4 RQ3: Effectiveness of Rule Extraction

In this section, we evaluate the effectiveness of VULGENIE in extracting API security rule using a dataset of 150 Java security patches (collection details are described in §4.1).

Ground Truth Construction. Two authors manually examined security patches to extract API security rules, merged

with VULGENIE's results. Rules were validated using disclosed or crafted payloads to ensure exploitability.

Result Overview. Table 4 summarizes the overall results. Across 150 Java security patches, VULGENIE successfully extracted 198 distinct API security rules. Evaluated against the ground truth, VULGENIE achieved 81.82% precision and 92.96% recall, indicating that its patch-guided mining recovers most SAPIs and their exploit conditions implicitly embodied in the patches while keeping false positives modest.

Notably, VULGENIE extracted 62 API security rules that were absent from manual analysis, demonstrating its ability to not only reduce analyst effort but also significantly enhance coverage. This is primarily because VULGENIE automatically tracks not only SAPIs in the patch modified method (like APHP) but also accurately identify inter-library SAPI calls via cross-validation and SAPI expansion strategy (in §3.2), that are extremely challenging to enumerate manually. For the motivating example, VULGENIE not only identified five SAPI calls in the diff code context in P_{sec}, but also uncovered ten SAPIs in JXPath sharing the same vulnerability root cause.

Table 4: Ablation study for two variants of VULGENIE (RQ3).

Tool	TP	FP	FN	Precision	Recall
VULGENIE	198 (177)	44	15	81.82%	92.96%
VULGENIE-NOMBG	33 (20)	65	180	33.67%	15.49%
VULGENIE-NOCROSSVAL	47 (19)	126	166	27.17%	22.07%

False Positive and Negative Analysis. Among the 44 false positives (FPs) generated by VULGENIE, the most common vulnerability types were XSS (16 FPs), path traversal (13 FPs), and XXE (8 FPs). A post-hoc audit shows that these FPs mainly occur during LLM-guided SAPI selection, where VULGENIE disambiguates among candidates left after cross-validation or introduced by backward SAPI expansion from the identified root cause. Among them, 29 cases arose because VULGENIE failed to retrieve source code for candidate APIs in some third-party libraries, depriving the LLM of critical semantic context and causing incorrect selections. We manually supplemented the missing method source code, VULGENIE then eliminated these FPs. The remaining 15 cases resulted from the LLM's limited domain knowledge.

VULGENIE builds its source code static analysis engine atop Joern, and the 15 false negatives mainly stem from two limitations of Joern's Java front end. First (five cases), Joern fails to model certain Java-specific constructs (e.g., static initializer blocks), leaving the associated violations and defenses absent from the code property graph. This prevents VULGENIE from linking violations/defenses to the relevant SAPIs. Second (ten cases), Joern provides insufficient support for Java dynamic features, e.g., reflection. This disrupted VULGENIE's tracing of data/control-flow from violations/defenses back to the relevant SAPIs, leading to missed rules.

Distribution of SAPIs. As shown in Figure 9, 177 of the 198 API security rules extracted by VULGENIE were not covered by CodeQL across all eight vulnerability types. These rules largely account for VULGENIE 's superior vulnerability detection capability compared to CodeQL (as discussed in §4.2) and can also enhance other existing vulnerability detection tools. This advantage primarily arises from VULGENIE 's patch-guided rule extraction, which automatically derives API security rules from newly disclosed security patches, allowing the tool to incorporate the latest vulnerability knowledge in a timely manner. In contrast, traditional static analysis tools such as CodeQL depend on manually curated rule sets, which are often delayed relative to recent vulnerability disclosures.

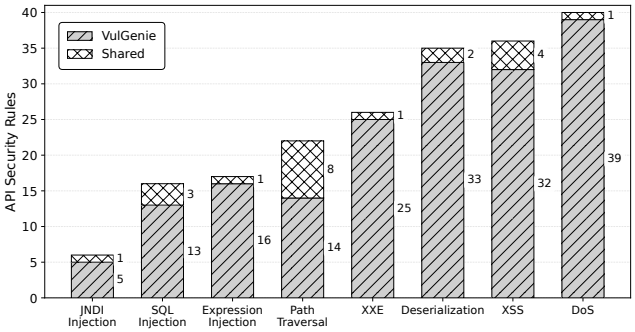
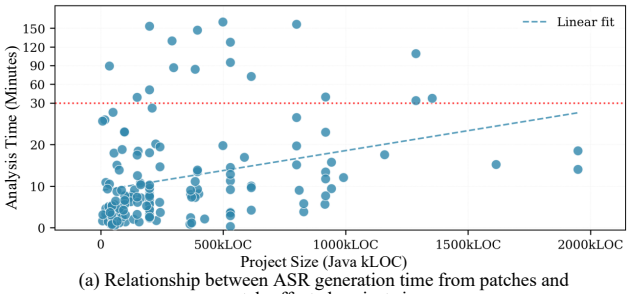


Figure 9: Distribution and Overlap of SAPIs Extracted by VULGENIE vs. CodeQL Across eight Vulnerability Types (Shared means SAPIs that are commonly owned)

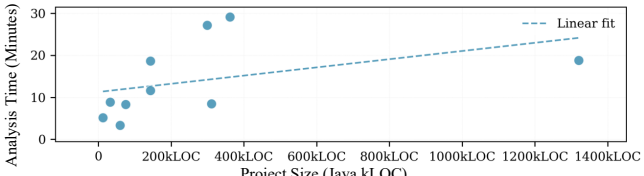
Ablation Study#2. We compare VULGENIE with two ablated variants to evaluate the contributions of its core components for API security rule extraction, as summarized in Table 4.

First, we evaluate the effectiveness of MBG-guided strategy by comparing VULGENIE with an ablated variant, VULGENIE-NOMBG, which disables the MBG-guided module and directly employs the LLM to select violations and defenses from the modified files in P_{sec}. VULGENIE-NOMBG achieves only 33 TPs with 33.7% precision and 15.5% recall, far below VULGENIE. This substantial performance drop highlights the critical role of MBG in consolidating security-relevant constraints and defenses dispersed across multiple files, thereby enabling accurate API security rule extraction.

Second, we evaluate the impact of the attack-defense cross-validation strategy by comparing VULGENIE with an variant VULGENIE-NOCROSSVAL, which disables the cross-validation and directly marks all traced APIs from violation-s/defense as security-sensitive. VULGENIE-NOCROSSVAL achieves 47 TPs, with a precision of 27.17% and a recall of 22.07%. The sharp precision drop confirms that cross-validation is vital for filtering spurious candidates and suppressing false positives, underscoring its central role in ensuring accurate and reliable SAPI identification.



(a) Relationship between ASR generation time from patches and patch-affected project size



(b) Relationship between vulnerability detection time and target project size.

Figure 10: Relationship between VULGENIE's analysis time and project size.

4.5 RQ4: Performance

In this section, we evaluate VULGENIE's performance overhead across two critical modules: (1) API security rule extraction (RQ3 in §4.4), and (b) vulnerability detection (RQ1 in § 4.2).

For rule extraction, it processes each patch in 22 minutes 39 seconds on average, using 14 LLM queries consuming 21,627 tokens per patch (224 diff lines average). Figure 10 (a) shows that VULGENIE 's rule extraction time grows roughly linearly with the size of the application fixed by the patch. Several outliers exist, such as some million-line projects that finish faster than smaller ones. This occurs because the analysis cost is also influenced by patch complexity, e.g., the number of modified lines. More complex patches can require additional LLM queries to identify defense-related variables and synthesize ASRs, and DeepSeek's slower responses further contribute to the cost. Overall, VULGENIE finishes rule extraction within 30 minutes for 133 out of the 150 patches, indicating its efficiency.

For vulnerability detection, VULGENIE analyzes an entire application (275,993 lines average) in an average of 13 minutes and 58 seconds. Considering the complexity of real-world Java patches and applications, we expect the analysis time to remain within acceptable and manageable limits. As shown in Figure 10 (b), VULGENIE 's detection time also increases roughly linearly with the size of the analyzed project. Some larger projects appear faster than smaller ones because VULGENIE detects only the code regions that are potentially relevant to the vulnerability rather than scanning the entire codebase. In addition, our intersection- and deviation-guided path- and context-sensitive strategy further eliminates unnecessary analysis time.

5 Discussion

Limitations. We discuss two limitations of VULGENIE. First, although VULGENIE extends Joern with a higher-precision interprocedural control- and data-flow analysis engine and can effectively handle certain dynamic features (e.g., polymorphism), it remains limited in some others (e.g., reflection), obscuring the SAPIs introduced or fixed in patches. Second, while VULGENIE reasons about some implicit constraints, other implicit conditions (e.g., external dependencies) must also be resolved to verify source-to-sink paths' exploitability.
Support for Complex Patches. Our empirical study shows that patches (in §4.2) modify an average of 224 lines of code. Such complex patches often faces numerous violation-irrelevant modifications, dispersed defense logic, and complex dependencies between modifications and SAPIs, making manual identification of SAPIs and their exploit conditions particularly challenging and time-consuming. However, existing tools (e.g., [24], [14]) typically overlook this common scenario, supporting rule extraction only from simpler, single-file patches (e.g., APHP supports patches with fewer than 20 modified lines). In contrast, VULGENIE leverages MBG-guided violation and defense identification combined with attack-defense cross-validation, enabling accurate extraction of API security rules even from complex, multi-file patches. These capabilities also allow VULGENIE to automatically and reliably infer the relevant SAPIs without relying on any prior SAPI knowledge.

6 Related Work

API document based approach. Documentation–driven detectors extract API-usage constraints from natural language artifacts like manuals to find violations [29,31,34]. This method provides extensive coverage, as every public method is described somewhere, they can generate rules even for APIs that have never been linked to a vulnerability. However, the specifications they rely on are written in natural language, which is inherently ambiguous, underspecified, and occasionally stale, so the inferred rules inherit those gaps. Lacking concrete exploit traces, these rules are often over-approximate and make it difficult to prioritize true positives during triage [6]. By contrast, VULGENIE mines rules solely from commits that repaired real exploits, guaranteeing that each API Security Rule is grounded in an attack that actually occurred.
Patch-based API-misuse rule extraction. Patch-oriented techniques mine API specifications from security fixes. APHP [24] isolates target APIs, post-operations, and critical variables to detect post-handling bugs in C code, mainly identifying memory leaks due to missing paired calls. Seal [14] builds pre-/post-patch program-dependence graphs from Linux kernel patches and lifts value-flow differences into reachability rules, but focuses on memory management and locking errors. Like APHP and Seal, VULGENIE derives specifications from

patches, but targets Java's distinct threat landscape: deserialization flaws, path-traversal vulnerabilities, command injections, which hinge on logical conditions rather than memory properties. Extracting these higher-level constraints requires decomposing multi-vulnerability patches into dependency-aware subgraphs and combining static taint analysis with LLM reasoning to recover the logical exploitation conditions that characterize modern Java API-misuse vulnerabilities.

Code clone-based and ML approaches. Clone-matching and machine-learning detectors [19, 28, 41] infer "correct" behaviour by mining large corpora and flagging deviations. For example, VUDDY employs a multi-level abstraction scheme for method-level clone detection [21], which relies on syntactic similarity. Conversely, VULGENIE learns vulnerabilities' root causes by inferring security-sensitive APIs and exploit conditions, even far from patch diffs, enabling cross-application generalization. APISAN extracts semantic beliefs from diverse projects using symbolic execution [43], GPTAid leverages LLMs to generate API-parameter security rules but suffers from brittleness across models [26], and FICS clusters API usage graphs to identify outliers [12]. These approaches assume frequent usage is correct, a heuristic that fails when erroneous patterns dominate codebases before fixes are applied. In contrast, VULGENIE learns directly from security patches, generating API Security Rules that encode taint and data-flow constraints and thereby sidestep the majority-vote fallacy while providing immediately actionable findings.

7 Conclusion

The paper proposes VULGENIE, a novel framework for Java API security rule (ASR) extraction and misuse vulnerability detection. VULGENIE combines MBG-guided violation and defense extraction with attack-defense cross-validated SAPI inference to accurately extract security rules, and applies intersection- and deviation-guided path- and context-sensitive taint analysis to detect zero-day vulnerabilities with high precision and efficiency.

We evaluated VULGENIE's performance for ASR extraction and vulnerability detection, demonstrating its effectiveness and scalability in real-world.

Acknowledgement

We would like to thank anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172104, 62172105, 62202106, 62302101, 62102091, 62472096, 62402114, 62402116), SNSF grant PCEGP2_186974, and ERC Horizon 2020 grant 850868. This work is sponsored by CAAI-Ant Group Research Fund. Lei Zhang is the corresponding author. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). We also thank Fute Sun for valuable discussions on static program analysis algorithms.

Ethical Considerations

Vulnerability Disclosure. Our evaluation was conducted with strict attention to security and responsible handling. We established a consistent disclosure process and privately reported all 46 identified vulnerabilities to the corresponding project maintainers (the key stakeholders), following their official disclosure guidelines. For example, vulnerabilities in Apache Seata and Apache IoTDB were reported through the ASF Vulnerability Reporting and Handling Process, while issues in Undertow were disclosed following the Red Hat Product Security process. For projects without a dedicated security policy, we submitted reports through the contact emails provided in their GitHub or Gitee repositories.

For each vulnerability report, we provided the maintainers with detailed technical information, including a proof-of-concept payload and a proposed remediation, enabling them to reproduce, verify, and fix the issue effectively. To prevent the exposure of sensitive information, we did not release any exploit payloads or other potentially harmful artifacts publicly at any point during this process. This approach ensures that our research contributes to improving software security without introducing unnecessary risks. In addition, we avoided disclosing any unpatched vulnerability details in the paper. Therefore, the publication of this paper does not pose any safety risk to real-world users.

Vulnerability Verification Environment. In our evaluation, all Java applications analyzed by VULGENIE are open-source. We downloaded these projects and performed static analysis on them locally using VULGENIE. After VULGENIE reported potential vulnerabilities, we built and executed the applications locally to verify the findings. Neither the static analysis nor the verification process involved any real user data or privacy-sensitive information.

Open science

To facilitate further research, we have publicly released the implementation of VULGENIE and the Java security patch dataset used in our evaluation on Zenodo [10], ensuring permanent availability.

References

[1] Codeql. https://codeql.github.com/.

[2] Diff tool. https://manpages.ubuntu.com/manpages/trusty/man1/diff.1posix.html.

[3] Geotool. https://geotools.org/.

[4] Jaccard index. https://en.wikipedia.org/wiki/Jaccard_index.

[5] Maven repository. https://mvnrepository.com/.

[6] The Midas Touch: Triggering the Capability of LLMs for RM-API Misuse Detection.

[7] networkx.org. 2024. networkx implementation of pagerank. https://networkx.org/documentation/networkx-1.10/reference/generated/networkx.algorithms.link_analysis.pagerank_alg.pagerank_scipy.html.

[8] Open-source code analysis platform for c/c++/java/binary/javascript/python/kotlin based on code property graphs. discord https://discord.gg/vv4mh284hc. https://github.com/joernio/joern.

[9] A security commit in qlexpress. https://github.com/alibaba/QLExpress/commit/b872fed71a4508fceffd02c70e79f36aeeb137bb.

[10] Source code of vulgenie. https://doi.org/10.5281/zenodo.17972521.

[11] Tai-e. https://github.com/pascal-lab/Tai-e.

[12] Mansour Ahmadi, Reza Mirzazade Farkhani, and Ryan Williams. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code.

[13] Bofei Chen, Lei Zhang, Xinyou Huang, Yinzhi Cao, Keke Lian, Yuan Zhang, and Min Yang. Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction. In 2024 IEEE Symposium on Security and Privacy (SP), pages 3961–3978, 2024.

[14] Wei Chen, Bowen Zhang, Chengpeng Wang, Wensheng Tang, and Charles Zhang. Seal: Towards diverse specification inference for linux interfaces from security patches. In Proceedings of the Twentieth European Conference on Computer Systems, pages 1246–1262, 2025.

[15] Convertigo. Convertigo is an open source low code platform including a no code application builder for full-stack mobile and web application development, 2025. Accessed: 2025-08-25.

[16] DeepSeek. Deepseek, unravel the mystery of agi with curiosity. answer the essential question with long-termism. https://www.deepseek.com/, 2024. Accessed: 2025-08-24.

[17] Dromara Community. Hutool: A set of tools that keep Java sweet, 2024. Accessed: 2025-01-18.

[18] Peiwei Hu, Ruigang Liang, Ying Cao, Kai Chen, and Runze Zhang. {AURC}: Detecting errors in program code and documentation. In 32nd USENIX Security Symposium (USENIX Security 23), pages 1415–1432, 2023.

[19] Jiasheng Jiang, Jingzheng Wu, Xiang Ling, Tianyue Luo, Sheng Qu, and Yanjun Wu. APP-Miner: Detecting API Misuses via Automatically Mining API Path Patterns. In 2024 IEEE Symposium on Security and Privacy (SP), pages 4034–4052, May 2024.

[20] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In Proceedings of the 31st IEEE/ACM international conference on automated software engineering, pages 472–482, 2016.

[21] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In 2017 IEEE Symposium on Security and Privacy (SP), pages 595–614, May 2017.

[22] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In International Conference on Compiler Construction, pages 47–64. Springer, 2006.

[23] Xia Li, Jiajun Jiang, Samuel Benton, Yingfei Xiong, and Lingming Zhang. A Large-scale Study on API Misuses in the Wild. In 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pages 241–252, April 2021.

[24] Miaoqian Lin, Kai Chen, and Yang Xiao. Detecting {API} {Post-Handling} Bugs Using Code and Description in Patches. In 32nd USENIX Security Symposium (USENIX Security 23), pages 3709–3726, 2023.

[25] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting missed security operations through differential checking of object-based similar paths. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 1627–1644, 2021.

[26] Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating API Parameter Security Rules with LLM for API Misuse Detection. In Proceedings 2025 Network and Distributed System Security Symposium, San Diego, CA, USA, 2025. Internet Society.

[27] Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. Proceedings of the ACM on Programming Languages, 3(OOPSLA):1–29, 2019.

[28] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting {Missing-Check} Bugs via Semantic- and {Context-Aware} Criticalness and Constraints Inferences. In 28th USENIX Security Symposium (USENIX Security 19), pages 1769–1786, 2019.

[29] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 1837–1852, Virtual Event USA, October 2020. ACM.

[30] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. Context sensitivity without contexts: A cut-shortcut approach to fast and precise pointer analysis. Proceedings of the ACM on Programming Languages, 7(PLDI):539–564, 2023.

[31] Yunlong Ma, Wentong Tian, Xiang Gao, Hailong Sun, and Li Li. API Misuse Detection via Probabilistic Graphical Model. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, pages 88–99, New York, NY, USA, September 2024. Association for Computing Machinery.

[32] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages 1–11, 2002.

[33] NIST National Vulnerability Database. CVE-2023-24163: SQL injection in Dromara hutool, 2023. CVSS Score: 9.8 (Critical).

[34] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. Api-misuse detection driven by fine-grained api-constraint knowledge graph. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pages 461–472, 2020.

[35] Yang Song, Xuzhou Zhang, and Yun-Zhan Gong. Infeasible path detection based on code pattern and backward symbolic execution. Mathematical Problems in Engineering, 2020(1):4258291, 2020.

[36] Shiyu Sun, Yunlong Xing, Xinda Wang, Shu Wang, Qi Li, and Kun Sun. Dispatch: Unraveling security patches from entangled code changes.

[37] Tree-sitter. Java grammar for tree-sitter. https://github.com/tree-sitter/tree-sitter-java/, 2022. Accessed: 2025-08-24.

[38] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In 2023 IEEE symposium on security and privacy (SP), pages 2658–2675. IEEE, 2023.

[39] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 149–160. IEEE, 2021.

[40] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exposing library api misuses via mutation analysis. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 866–877. IEEE, 2019.

[41] Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs. In Proceedings of the ACM Web Conference 2022, WWW '22, pages 755–766, New York, NY, USA, April 2022. Association for Computing Machinery.

[42] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In 29th USENIX Security Symposium (USENIX Security 20), pages 2397–2414, 2020.

[43] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. {APISan}: Sanitizing {API} usages through semantic {Cross-Checking}. In 25th USENIX Security Symposium (USENIX Security 16), pages 363–378, 2016.

[44] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. An empirical study on api parameter rules. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pages 899–911, 2020.

[45] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 27–37. IEEE, 2017.

A Design Details

Table 5 illustrates the categorization of repair pattern types. Algorithm 2 illustrates the MBG-guided SAPI-specific patch isolation approach.

Table 5: The categorization of repair pattern types.

Type	Description
Modify Variable Values	(change/add/delete) the value of the variable to change the program state
Modify Variable Declaration	(change/add/delete) the declaration/definition statements of a variable/field to modify it's properties, e.g., access modifier
Modify Class Declaration	(change/add/delete) the declaration/definition statements of the class to modify it's properties
Modify Method Declaration	(change/add/delete) the declaration/definition statements of the method to modify it's properties
Modify Method Call	(change/add/delete) the invocation of a method, and optionally replace it with a safer alternative by adjusting its properties (e.g., method name, parameters, or return type) to strengthen program security
Modify Error Handling	(change/add/delete) error/exception handling code (approaches to managing errors/exceptions within the program flow)
Modify Sanity Check	(add/change/delete) a sanity check (on variable) to verify the security of a specific program state (by invoking a method, assertion, etc.)
Move Statements	move the position of statements (in the context without modification)
In loop	(add/change/delete) a loop structure or relocate existing statements into/out of a loop, thereby modifying the iteration logic (e.g., ensuring a check is applied to each element, restricting the number of iterations, or avoiding redundant executions)
Others	no template: uncommon minor changes that cannot be categorized into any of the above types

Algorithm 2 SMBG Isolation Algorithm

```
function ConstructSMBGs(MBG)        ▷ Input: Modification Behavior Graph MBG; Output: SAPI-specific subgraphs SMBGs
    SMBGs ← []
    vertexUnits ← getVertexUnits(MBG)
    for vertexUnit ∈ vertexUnits do
        SMBG ← []
        visited ← []
        stack.push(vertexUnit)
        while stack ≠ ∅ do
            unit ← stack.pop()
            if unit ∈ visited then
                continue
            end if
            visited.add(unit)
            SMBG.add(unit)
            depUnits ← getDependencyUnits(MBG, unit)
            for depUnit ∈ depUnits do
                if haveNoSemanticEqual(visited, depUnit) then
                    stack.push(depUnit)
                end if
            end for
        end while
        if isNotDuplicate(SMBGs, SMBG) then
            SMBGs.add(SMBG)
        end if
    end for
    return SMBGs
end function
```