

# Every Scenario Matters: Scenario-Guided White-Box Testing for Static Code Checkers Powered by LLMs

Jun Liu

School of Advanced Interdisciplinary  
Sciences  
University of Chinese Academy of  
Sciences  
Beijing, China  
Institute of Software, Chinese  
Academy of Sciences  
Beijing, China  
[liuj2022@ios.ac.cn](mailto:liuj2022@ios.ac.cn)

Linjie Pan

Jinsong Ren

[panlinjie@huawei.com](mailto:panlinjie@huawei.com)  
[renjinsong@huawei.com](mailto:renjinsong@huawei.com)  
Huawei Technologies Co., Ltd.  
Beijing, China

Liqiang Ji\*

Key Laboratory of System Software  
(Chinese Academy of Sciences)  
Institute of Software, Chinese  
Academy of Sciences  
Beijing, China  
[jilq@ios.ac.cn](mailto:jilq@ios.ac.cn)

Jun Yan\*

Beijing Key Laboratory of Intelligent  
Software Engineering  
Beijing, China  
Key Laboratory of System Software  
(Chinese Academy of Sciences)  
Institute of Software, Chinese  
Academy of Sciences  
Beijing, China  
[yanjun@ios.ac.cn](mailto:yanjun@ios.ac.cn)

Jiwei Yan<sup>✉\*</sup>

Beijing Key Laboratory of Intelligent  
Software Engineering  
Beijing, China  
Institute of Software, Chinese  
Academy of Sciences  
Beijing, China  
[yanjiwei@otcaix.iscas.ac.cn](mailto:yanjiwei@otcaix.iscas.ac.cn)

Jian Zhang<sup>✉\*</sup>

Key Laboratory of System Software  
(Chinese Academy of Sciences)  
Institute of Software, Chinese  
Academy of Sciences  
Beijing, China  
[zj@ios.ac.cn](mailto:zj@ios.ac.cn)

## Abstract

In static analysis tools, each detection rule corresponds to a dedicated code checker. The checker's quality directly determines the validity of reported defects. Thus, comprehensive testing of these checkers is crucial. Existing checker testing approaches usually generate test cases from high-level rule descriptions while treating checkers as black boxes. However, these descriptions usually omit implementation-level conditions and checking scenarios. To bridge this gap, *our key insight* is that the rule description and checker implementation provide complementary information for comprehensive testing. The implementation supplies concrete checking logic, while the rule anchors the intended semantics.

Building on this, we propose SCENSCT, a novel scenario-guided white-box checker testing approach powered by LLMs. SCENSCT first translates the checker's source code into a unified logic-based representation (CheckerIR) and applies both *logic-decomposition* and *scenario-validation strategies* to extract concrete checking scenarios. It then converts these scenarios into test cases through *execution-feedback-guided test alignment* to instantiate scenario constraints and *scenario-invariant test augmentation* to generate variants that preserve the target scenario. We evaluate SCENSCT

on real-world Java checkers from five mainstream static analysis tools. In single-tool evaluation, SCENSCT extracts an average of ~9 scenarios and generates ~39 tests per checker, uncovering 270 potential issues (58 false positives and 212 false negatives) with an overall precision of 86.4%. Furthermore, cross-tool evaluation reveals 177 capability differences among shared checkers. Overall, SCENSCT offers a systematic testing method to improve the reliability of static analysis tools.

## CCS Concepts

• **Software and its engineering** → **Software reliability**.

## Keywords

Static Checker Testing, Test Generation, Large Language Models

### ACM Reference Format:

Jun Liu, Liqiang Ji, Jiwei Yan, Linjie Pan, Jinsong Ren, Jun Yan, and Jian Zhang. 2026. Every Scenario Matters: Scenario-Guided White-Box Testing for Static Code Checkers Powered by LLMs. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3832783.3834381>

\*Also with University of Chinese Academy of Sciences.



This work is licensed under a Creative Commons Attribution 4.0 International License. ASE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-2882-2/2026/10  
<https://doi.org/10.1145/3832783.3834381>

## 1 Introduction

Static analysis tools [37] are widely used in modern software development to detect bugs, issues, and defects [38, 48]. By scanning source code without actually executing it, they can identify potential problems early in the development process [7]. Modern static

analysis tools now support many programming languages and detect a broad range of code anomalies [3, 23]. As these tools are increasingly used in software quality assurance, a fundamental question arises: *how reliable are the tools themselves?*

A static analysis tool comprises a set of rule checkers (or simply, *checkers*) [19, 37]. Each checker is a basic analysis unit that detects a specific type of code anomaly based on a general rule. Hence, the precision and soundness of individual checkers largely determine the overall reliability of the tool [56]. An imprecise checker produces false positives (FPs), increasing developers' manual verification burden, while an unsound one yields false negatives (FNs), allowing real defects to go undetected. Even though existing checkers are typically carefully designed and implemented, they still suffer from these issues. Recent empirical studies [13, 44] show that FPs and FNs are prevalent throughout the development lifecycle of static analysis tools. User surveys further confirm that high FP and FN rates remain a major barrier to broader industrial adoption of these tools [12, 25, 38]. Therefore, systematically testing checkers to uncover precision and soundness defects is essential [47, 56].

Despite its importance, testing a static checker remains challenging because it often encodes multiple checking scenarios, including boundary conditions omitted from high-level rule descriptions. Existing approaches are essentially black-box and therefore struggle to cover these scenarios. One line of work tests the tool as a whole using real-world or manually crafted programs [14, 19, 27, 47, 50]. Without targeting individual checkers, these approaches leave many checkers untested. For example, Wang et al. [50] tested only 74 of 545 SonarQube checkers. Another line of work targets individual checkers but still treats the implementation as a black box, relying solely on abstract rule descriptions and limited official examples [40, 56]. As a result, the generated tests rarely exercise the implementation-level scenarios encoded by a checker. To characterize a checker's capability boundary, *every scenario matters*: each checking scenario requires tests that instantiate its conditions.

To address this, we need to answer a core question: *how can we identify the checking scenarios a checker is designed to evaluate?* Our key insight is that the rule description and the checker implementation are complementary for scenario extraction. The **rule description** conveys the intended semantics, while the **checker implementation** encodes concrete checking conditions. We first derive fine-grained scenarios from the implementation, then validate their semantics and expected outcomes against the rule. To enable this white-box scenario extraction and test generation pipeline, we must address two main challenges:

- C1: Extracting intended checking scenarios.** Checkers from different tools follow different implementation paradigms and rely on complex, framework-specific APIs. The structural diversity and implementation complexity make it difficult to universally parse checker code and extract fine-grained checking scenarios with intended semantics.
- C2: Converting scenarios into effective test cases.** Directly generated tests may omit constraints of the intended scenario. Unless a test is shown to instantiate its scenario, a disagreement cannot be attributed to checker behavior. Moreover, one test cannot exercise the diverse code forms that may preserve the same scenario.

To address these challenges, we propose **SCENSCT**, a novel scenario-guided white-box approach for testing checkers via LLMs. To extract intended scenarios (**C1**), SCENSCT introduces **Logic-Based Scenario Extraction**. We design CheckerIR, a unified and grammar-constrained domain-specific language that declaratively models a checker's checking conditions in first-order logic. SCENSCT translates checker code into CheckerIR and extracts intended positive and negative scenarios through *logic-decomposition* and rule-anchored *scenario-validation* strategies. To convert these scenarios into effective tests (**C2**), SCENSCT introduces **Scenario-Guided Test Generation**. For each scenario, it generates a seed and applies *execution-feedback-guided alignment* to repair generation errors under the scenario's fixed constraints, using actual checker execution as observed feedback. If the test satisfies the scenario but the observed checker result still disagrees with this expected outcome, SCENSCT reports it as a candidate FP/FN. Aligned seeds then undergo *scenario-invariant augmentation* to produce diverse variants that preserve the target scenario.

We implement SCENSCT and evaluate it on Java code checking<sup>1</sup>, as Java remains one of the two most widely used programming languages [1]. We evaluate SCENSCT in two settings, *single-tool testing* and *cross-tool testing*, using real-world checkers from five mainstream tools (CodeQL [3], PMD [5], Checkstyle [2], Error Prone [4], and SonarQube [6]). In single-tool testing on 89 defect-related CodeQL checkers, SCENSCT extracts ~9 scenarios and generates ~39 tests per checker on average. Manual assessment identifies 311 FP/FN cases among 360 candidates (86.4% precision), leaving 270 unique cases after deduplication (58 FPs and 212 FNs). Cross-tool testing further reveals 177 cases that characterize capability differences among shared security-related checkers across the five tools. Overall, SCENSCT demonstrates its effectiveness in systematically testing static checkers. We summarize our contributions as follows:

- We propose a novel scenario-guided white-box testing approach for checkers via LLMs. To our knowledge, it is the first work on white-box checker testing that synergizes the checker's rule and implementation.
- We introduce a logic-based scenario extraction method by designing a unified domain-specific language (CheckerIR) to model checker code and using logic-decomposition and scenario-validation strategies to extract intended scenarios.
- We develop a scenario-guided test generation method to convert scenarios into effective tests with execution-feedback-guided alignment and scenario-invariant augmentation.
- We implement our approach as SCENSCT and evaluate it across five mainstream static checking tools. Both the single-tool and cross-tool evaluation results show that SCENSCT is effective in systematically testing individual checkers.

## 2 Background & Motivation

### 2.1 Static Code Checker

Static code checkers are the fundamental units of static analysis tools [36]. Each checker enforces a specific rule to detect a particular type of code anomaly [24, 50]. Although rules are often described in general natural language, checker implementations are often

<sup>1</sup>SCENSCT is designed to be language-agnostic and readily extensible to other languages. See detailed explanations in Section 5.

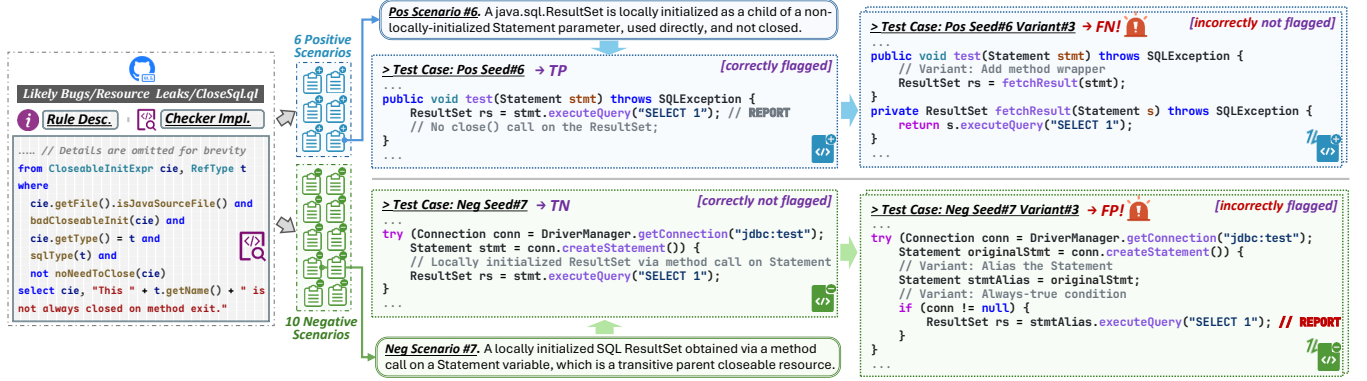


Figure 1: A motivating example showing that SCENSCT extracts 16 scenarios from CodeQL’s checker CloseSql.q1 and generates metamorphic variants to successfully identify a false positive (FP) and a false negative (FN) in CodeQL.

complex and vary across tools. They may rely on different program representations, such as Abstract Syntax Trees (ASTs), Data Flow Graphs (DFGs), or Control Flow Graphs (CFGs), and employ diverse analysis algorithms, such as AST-based pattern matching and data flow analysis [32, 37]. Moreover, checkers can be expressed in different forms, e.g., CodeQL uses QL queries, while PMD uses Java or XML to implement checkers. Even for the same rule, the checkers’ implementations differ because of differences in framework-specific APIs and underlying analysis mechanisms. This diversity makes it difficult to universally parse and analyze checker code. To bridge this gap, our approach introduces *CheckerIR*, a unified domain-specific language that models diverse checkers, enabling a general pipeline for scenario extraction and test generation.

## 2.2 Task Definition

In static checker testing, a *test case*  $t \in T$  is an input program used to assess whether a checker implementation  $I$  correctly enforces its target rule  $\mathcal{R}$ . We divide the test-case space  $T$  into two disjoint sets: a positive set  $T^+$  and a negative set  $T^-$ . A positive test case  $t \in T^+$  violates  $\mathcal{R}$  and is expected to trigger an alert, whereas a negative test case  $t \in T^-$  complies with  $\mathcal{R}$  and is expected to remain silent. For each test case  $t \in T$ , executing  $I$  yields a binary decision  $D_I(t) \in \{0, 1\}$ , where  $D_I(t) = 1$  denotes that an alert is emitted and  $D_I(t) = 0$  denotes no alert. Comparing this decision with the expected outcome yields four evaluation categories:

- **True Positives:**  $TP = \{t \in T^+ \mid D_I(t) = 1\}$
- **True Negatives:**  $TN = \{t \in T^- \mid D_I(t) = 0\}$
- **False Positives:**  $FP = \{t \in T^- \mid D_I(t) = 1\}$  (false alarms)
- **False Negatives:**  $FN = \{t \in T^+ \mid D_I(t) = 0\}$  (missed alarms)

As introduced in Section 1, our task focuses on *scenario-guided checker testing*. Unlike an individual low-level condition, a *checking scenario*  $s$  combines related implementation conditions into a concrete, realizable detection path. Each scenario is validated against  $\mathcal{R}$  and assigned an expected polarity: a positive scenario ( $s \in S^+$ ) violates the rule and should trigger an alert, whereas a negative scenario ( $s \in S^-$ ) complies with the rule and should remain silent.

First, we extract these scenarios from  $I$  and  $\mathcal{R}$ :

$$\text{Extract}(I, \mathcal{R}) = S^+ \cup S^- \quad (1)$$

Second, for each scenario  $s$ , we generate test cases that instantiate it by satisfying its encoded constraints:

$$\text{Generate}(s) = T_s = \{t \in T \mid t \text{ satisfies } s\}. \quad (2)$$

The rule-validated scenario determines each test’s expected outcome. In contrast,  $D_I(t)$  is the observed checker behavior. Comparing the two yields the *TP*, *TN*, *FP*, and *FN* categories above.

## 2.3 Motivating Example

To illustrate the motivation of SCENSCT, we use the Java CodeQL checker CloseSql.q1 (located in “Likely Bugs/Resource Leaks/”) as a real-world example. We choose CodeQL due to its widespread adoption and effectiveness, as it supports over ten major programming languages and serves as the foundation of GitHub Advanced Security [3, 46].

The rule description of the checker states: “A database resource in the java.sql package that is opened but not closed may cause a resource leak”. For this rule, CodeQL’s developers only provide two official examples: (1) a *positive* pattern that allocates java.sql.Statement and java.sql.ResultSet resources without closing them, and (2) a *negative* pattern that uses a safe try-with-resources block for the database resources. These examples cover only a small part of the checking scenarios encoded in the implementation. We also asked recent LLMs to generate comprehensive positive and negative tests using only the rule description. Across three runs, the generated tests covered at most 6 of the 16 checking scenarios in Figure 1. They missed scenarios such as Pos Scenario #6, which initializes a resource through parameter passing. Thus, neither the official examples nor rule-only generation provides comprehensive scenario coverage, motivating our white-box approach.

To address this limitation (Challenge C1), SCENSCT translates checker implementations into CheckerIR and extracts checking scenarios through logic-based decomposition and rule-based validation. As shown in the left part of Figure 1, SCENSCT extracts **six** valid positive scenarios and **ten** valid negative scenarios. For example, Pos Scenario #6 details a ResultSet initialized from a

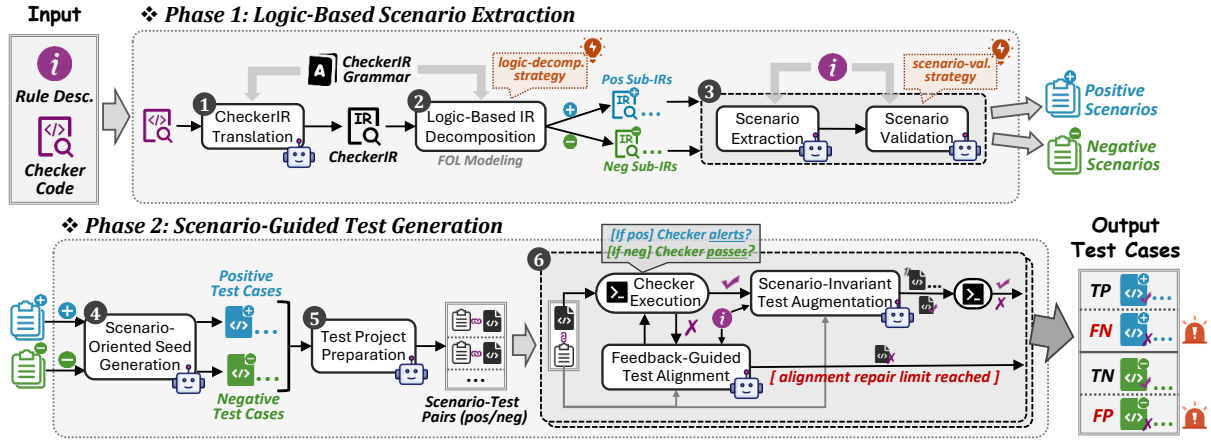


Figure 2: Overview of SCENSCT.

Statement parameter that is never closed, whereas *Neg Scenario #7* describes a *ResultSet* obtained from a local *Statement* variable acting as a transitive parent closeable resource.

The second challenge (C2) involves translating these extracted scenarios into test cases that instantiate their constraints. To overcome this, SCENSCT employs scenario-guided test generation by generating an aligned seed test for each scenario and applying scenario-invariant augmentations to create diverse variants. The right part of Figure 1 demonstrates how this approach exposes defects. For *Pos Scenario #6*, while the seed test (*Pos Seed#6*) properly triggers an alarm, its method-wrapper variant (*Pos Seed#6 Variant#3*) evades the checker’s static tracking, isolating an FN. Conversely, for *Neg Scenario #7*, its seed test (*Neg Seed#7*) safely passes, but the semantics-preserving variant (*Neg Seed#7 Variant#3*) incorrectly triggers a leak alert, exposing an FP caused by incomplete alias and flow resolution logic. These findings validate the effectiveness of SCENSCT in uncovering precision and soundness defects in static code checkers.

### 3 Methodology

In this section, we present the methodology of SCENSCT in two phases: scenario extraction (Section 3.2) and test generation (Section 3.3), after an overview in Section 3.1. The CodeQL CloseSql.q1 checker introduced in Section 2.3 serves as a running example.

#### 3.1 Overview

As shown in Figure 2, SCENSCT takes a checker’s code and its rule description as input and automatically produces a labeled test suite that exposes potential FPs and FNs. It operates in two phases:

➤ **Phase 1: Logic-Based Scenario Extraction.** To handle the diversity of checker implementations, SCENSCT first translates the given checker code into a unified, grammar-constrained logic representation called *CheckerIR* ①. Based on formal first-order logic modeling, it then disjunctively decomposes this global *CheckerIR* into positive and negative sub-clauses (pos/neg Sub-IRs) with the proposed logic-decomposition strategy ②. Finally, guided by the rule description, SCENSCT extracts and validates each scenario with

the designed scenario-validation strategy to finalize a set of discrete, valid positive and negative checking scenarios ③.

➤ **Phase 2: Scenario-Guided Test Generation.** SCENSCT first generates a seed test case for each scenario ④ and prepares an isolated project to organize and compile all seeds ⑤. It then refines each seed against its scenario and compares the observed checker result with the scenario’s fixed polarity. Execution feedback may repair generation errors, but cannot change the scenario or its expected outcome. SCENSCT next applies scenario-invariant augmentations to aligned seeds and reports residual disagreements as FP/FN candidates ⑥.

#### 3.2 Phase 1: Logic-Based Scenario Extraction

In this phase, SCENSCT synergizes the checker implementation and the rule description to extract discrete, valid checking scenarios.

**3.2.1 CheckerIR Translation ①.** As introduced in Section 1, we design *CheckerIR* to serve as a declarative domain-specific language that represents diverse checker implementations in a unified and grammar-constrained format, ready for first-order logic modeling. Checker implementations include query languages, source code, and declarative rules such as XML, so a deterministic translator would require tool-specific parsers and translation rules. SCENSCT instead uses an LLM for translation, while downstream parsing and decomposition are deterministic. Specifically, the grammar of *CheckerIR* is shown in Figure 3, where we reuse the fundamental logical connectors in first-order logic to bind sequences of atomic natural language conditions.

*CheckerIR* translation uses a three-step pipeline rather than one-shot generation. By separating logic transcription, logic refinement, and syntax correction, each step focuses on one requirement of a usable *CheckerIR*: checking-logic coverage, manageable structure, and grammar validity.

➤ **Logic Transcription:** We design transcription to retain report-triggering conditions. The prompt defines symbol and connector mappings with concise examples, then requires the LLM to preserve branch polarity, helper predicates, and control/data-flow constraints. Because helper methods may hide checking conditions,

```

check ::= condition    ▶ Root condition is typically EXISTS/OR
condition ::= atomic
            | AND { conditions }
            | OR { conditions }
            | NOT { condition }
            | EXISTS ( nodes ) { condition }
            | FORALL ( nodes ) { condition }
conditions ::= condition
            | condition, conditions
nodes ::= text          ▶ Identifier formatted as "type + unique ID"
            | text, nodes
atomic ::= ( text )     ▶ Single indivisible checking fact

```

**Figure 3: Concrete syntax of CheckerIR. Here, *text* denotes a backtick-delimited natural-language phrase.**

```

EXISTS ( ^CloseableInitExpr CIE^, ^RefType T^ ) {
  AND {
    ( ^CIE's type is T^ ),
    EXISTS ( ^RefType SUP^ ) {
      AND {
        ( ^SUP is an ancestor type of T^ ),
        ...
      }
      OR {
        ( ^SUP's qualified name is "java.sql.Connection" ),
        ( ^SUP's qualified name is "java.sql.Statement" ),
        ( ^SUP's qualified name is "java.sql.ResultSet" )
      }
    }
  }
}

```

**Figure 4: An example CheckerIR snippet for CloseSql.q1.**

the LLM recursively expands their logic instead of replacing it with abstract summaries. This step produces a draft CheckerIR for later refinement.

✎ **Logic Refinement:** The LLM is then instructed to refine the draft without changing its checking conditions. It removes residual non-checking noise, repairs malformed local structures such as missing wrappers or dummy blocks, and bundles sibling atomic conditions on the same element or property when a block becomes overly dense. This step bounds the structural complexity of CheckerIR for logic decomposition.

✎ **Syntax Correction:** Finally, SCENSCT guarantees formal well-formedness through a parser-guided repair loop. We parse the refined CheckerIR against the grammar in Figure 3 and feed concrete parser errors back to the LLM whenever validation fails. Since this stage is restricted to syntax repair, the LLM is explicitly instructed to only fix the syntax errors without changing the logic. This loop continues until a valid CheckerIR is obtained.

**Example.** Consider the CodeQL CloseSql.q1 checker from Section 2.3. After the three-step translation process, SCENSCT converts its QL source code into a CheckerIR. Figure 4 shows a snippet of the result. In this snippet, the checker requires the resource target to be a subtype of a SQL connection, statement, or result set.

**3.2.2 Logic-Based IR Decomposition** ②. Although CheckerIR captures the full checker logic, it is still too large to guide test generation directly. We therefore decompose the global CheckerIR into smaller Sub-IRs, where each Sub-IR isolates one logic path that can

### Algorithm 1 Logic Decomposition from CheckerIR to Sub-IRs

**Input:**  $ir$ : the full CheckerIR,  $cfg$ : decomposition configuration  
**Output:**  $S^+, S^-$ : the extracted positive and negative Sub-IR sets

```

1:  $S^+ \leftarrow \emptyset, S^- \leftarrow \emptyset$ 
2:  $Mvar, Msym, E \leftarrow parseForLogicModeling(ir)$ 
   ▶ parse CheckerIR into a logic formula with variable and symbol maps
3:  $G_{conf} \leftarrow identifyConflicts(Msym)$  ▶ identify conflict groups
4: for each  $(E_{cur}, S_{cur}) \in \{(E, S^+), (\neg E, S^-)\}$  do
5:    $C \leftarrow decompWithStrategy(E_{cur}, G_{conf}, cfg.maxDepth, cfg.combStrategy)$ 
   ▶ decompose with conflict-aware pruning and complexity bounding
6:   for each valid clause  $c \in C$  do
7:      $S_{cur}.add(reconstructSubIR(c, Mvar, Msym))$ 
   ▶ reconstruct each valid clause into a standalone Sub-IR
8:   end for
9: end for

```

later be translated into a concrete checking scenario. To preserve the original logical relations, SCENSCT first models CheckerIR as a first-order logic formula and then applies deterministic logic decomposition with conflict pruning and granularity control. The unified logic form enables the same procedure across tools. Algorithm 1 summarizes the high-level process.

Given a full CheckerIR, SCENSCT first maps quantifier-scoped nodes to logical variables and maps each atomic condition to a predicate symbol, yielding a logic formula that preserves the original checker structure (Line 2).<sup>2</sup> It next identifies conflict groups for contradiction pruning (Line 3) and decomposes both the original formula and its negated counterpart in the same loop (Lines 4–5). During decomposition (Line 5), SCENSCT follows the standard logic splitting procedure [8, 18]: it pushes negations inward and splits OR branches into conjunctive clauses. This process yields Sub-IRs that cover both positive logic paths that trigger reports and negative logic paths that represent compliant behavior. Each valid clause is finally reconstructed into a standalone Sub-IR (Lines 6–7).

*Logic-Decomposition Strategy.* During this process, the main challenge is not converting the logic itself but splitting a large formula into useful scenario-sized fragments without introducing impossible combinations or overwhelming expansion. We address this challenge with three strategy components:

- ★ **Conflict-aware pruning.** Before the decomposition, SCENSCT identifies mutually exclusive atomic conditions and records them as conflict groups. Any generated clause that contains an explicit contradiction (e.g., a clause containing  $P$  and  $\neg P$ ) or combines conditions from the same conflict group is discarded.
- ★ **Complexity bounding I: mutant aggregation.** As illustrated by SUP's three qualified-name alternatives in Figure 4, some disjunctive branches share the same predicate structure and differ only in the final concrete value. SCENSCT groups such branches into one atomic unit during decomposition, preventing multiplicative expansion from semantically similar alternatives while preserving the original disjunctive semantics.
- ★ **Complexity bounding II: hyperparameter setting.** SCENSCT further bounds decomposition complexity with two hyperparameters. First, it groups nested sub-expressions whose depth exceeds  $maxDepth$ <sup>3</sup>, so that low-level inner logic is not split too aggressively. Second, when multiple OR branches must be

<sup>2</sup>We implement an ANTLR-based parser to convert CheckerIR into the equivalent NLTK [9] first-order logic expression.

<sup>3</sup>Depth is measured by counting AND/OR branching connectors on the path to the root.

combined, SCENSCT employs combinatorial testing strategies (i.e.,  $t$ -way combinations [28, 52]) to limit clause generation, where `combStrategy` determines the combination strength. Together, `maxDepth` and `combStrategy` control the granularity of the resulting Sub-IRs. Lower `maxDepth` and weaker combinations result in fewer, coarser Sub-IRs, whereas higher values and stronger combinations yield more fine-grained ones.

**Example:** For the CheckerIR of `CloseSql.q1`, the three alternatives on SUP’s qualified name are aggregated as mutants during decomposition and preserved in the resulting Sub-IRs. SCENSCT further decomposes the CheckerIR into 7 positive and 16 negative Sub-IRs with `maxDepth` = 3 and `combStrategy` = 2-way.

**3.2.3 Scenario Extraction and Validation ③.** The decomposed Sub-IRs isolate distinct logic paths, but their symbolic form is still not directly suitable for test generation. SCENSCT therefore prompts the LLM to translate each Sub-IR into a concrete scenario description that preserves its detailed semantics, including the participating program elements, key relations, and expected behavior. These descriptions make the Sub-IR constraints usable for scenario validation and test generation. This extraction step is driven primarily by the Sub-IR itself, while the rule description is provided only as high-level contextual guidance.

**Scenario-Validation Strategy.** However, scenarios extracted from implementation logic alone may still deviate from the intended rule semantics: logic decomposition can still yield partially meaningless Sub-IRs, and the checker implementation itself may encode buggy logic. To address this, we propose a rule-anchored scenario-validation strategy. Instead of trusting the extracted scenario directly, SCENSCT uses the checker’s rule description as a semantic anchor to validate whether the scenario is realizable, non-redundant, and correctly labeled. Concretely, SCENSCT asks the LLM to validate each extracted scenario along three dimensions:

- ★ **Feasibility:** the scenario must be realizable as valid code under the target programming language semantics.
- ★ **Distinctness:** the scenario must represent a semantically distinct behavior rather than a duplicate.
- ★ **Polarity Consistency:** the scenario must agree with its positive/negative label under the rule description.

After validation, infeasible and duplicate scenarios are discarded, while polarity-inconsistent ones are reassigned to the correct group. The remaining validated scenarios serve as the final testing targets.

**Example:** For `CloseSql.q1`, SCENSCT extracts scenarios from 7 positive and 16 negative Sub-IRs. Rule-anchored validation prunes 1 duplicate from the positive set, yielding 6 finalized positive scenarios. For negatives, it removes 1 duplicate and 3 infeasible cases. It then randomly drops 2 additional scenarios to control testing overhead,<sup>4</sup> leaving 10 finalized negative scenarios.

### 3.3 Phase 2: Scenario-Guided Test Generation

This phase turns the extracted scenarios into compilable tests that instantiate their constraints, then creates diverse variants that preserve the same scenarios.

<sup>4</sup>In our implementation, we randomly select up to 10 positive and 10 negative scenarios per checker to control downstream execution costs, as detailed in Section 4.1.

**3.3.1 Scenario-Oriented Seed Generation ④.** Each scenario provides a concrete behavioral target for test generation. Given a scenario and its expected polarity, SCENSCT prompts the LLM to synthesize one minimal test case that instantiates the target behavior without introducing irrelevant code paths. The LLM is required to preserve the scenario’s concrete semantics, satisfy the intended positive/negative outcome, and keep the seed as small and local as possible. This step yields one focused seed test case per scenario as the starting point for downstream refinement and augmentation.

**3.3.2 Test Project Preparation ⑤.** To ensure that seed tests are valid and ready for checker evaluation, SCENSCT organizes all tests for a given checker in an isolated project with a unified layout. It then validates the project and iteratively repairs syntax and build errors while preserving the intended scenarios. Only tests that pass project validation proceed to checker execution.

Here, a key challenge is unresolved third-party dependencies, which pure LLM-based repair often fails to address when missing libraries cause compilation failures. To handle this, SCENSCT applies *static mock repair*. It first parses all the test cases to identify external libraries and collects the referenced classes. It then prompts the LLM to synthesize lightweight mocks only for the test cases that need them, following the rule of preserving exact API signatures while keeping bodies empty. This approach is effective for two reasons: recovering real dependencies is often impractical due to version ambiguity or unavailability, and static checkers typically depend only on signatures rather than library behavior, so signature-faithful mocks retain the checker-relevant semantics. As shown in Figure 5, this strategy enables SCENSCT to recover a buildable project even when the seed test requires external libraries.

**Example:** For `CloseSql.q1`, SCENSCT organizes all 16 seed tests into an isolated Java project and successfully compiles them with the mocked `HttpServletResponse` class.

**3.3.3 Scenario-Guided Test Alignment and Augmentation ⑥.** Prepared test cases are compilable, but they may still omit scenario constraints. SCENSCT therefore refines them under these constraints using checker execution as feedback.

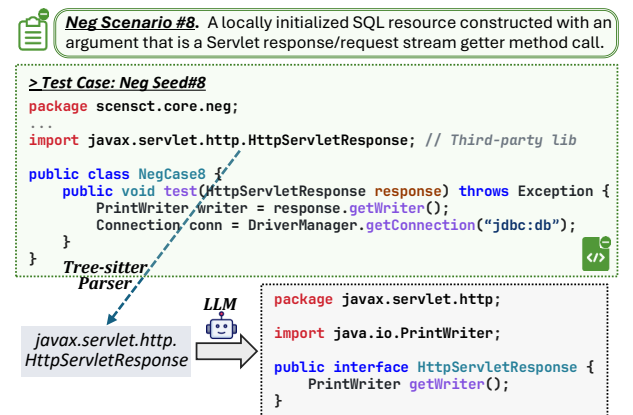


Figure 5: A complete negative test case for `CloseSql.q1` as well as the mocked `HttpServletResponse` class.

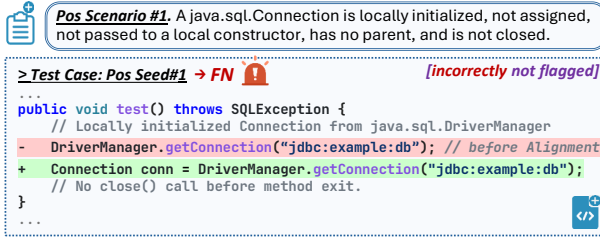


Figure 6: An FN candidate from the *Pos#1* scenario for *CloseSql.q1* that remains misaligned after refinement.

*Execution-Feedback-Guided Alignment.* The checker analyzes the prepared test project. The rule-validated scenario and its polarity define the expected behavior. The checker output (an alert or silence) is only observed feedback. On a mismatch, the LLM may repair generation errors but must preserve the scenario constraints and polarity. Within the configured budget, SCENSCT either obtains an aligned seed that instantiates the scenario and matches its polarity, or retains the unresolved disagreement as an FP/FN candidate.

**Example:** For *CloseSql.q1*, all seed tests except the one from *Pos#1* are aligned successfully. As shown in Figure 6, the generated *Pos#1* seed correctly instantiates the target scenario but consistently passes without an alert. SCENSCT therefore retains it as an FN candidate for the missed local unassigned initialization.

*Scenario-Invariant Augmentation.* Only aligned test cases proceed to augmentation. To exercise more code forms within the same checking scenario, SCENSCT uses an LLM to transform each test under a **scenario-invariant** constraint. The syntax and structure may change, but the target checking scenario must remain intact. Compared with fixed semantics-preserving transformations [11, 34, 56], this design can cover API-specific patterns and coding practices while keeping the scenario as the test target. Guided by common false-positive and false-negative patterns in static analysis tools [13, 44], we design four transformation classes:

- ★ **Lexical Refactoring:** Equivalent expression rewrites, declaration reshaping, and local variable renaming.
- ★ **Intra-Procedural Refactoring:** Introducing aliases or temporary variables, and reshaping local control flow.
- ★ **Inter-Procedural Refactoring:** Extracting helpers, inlining methods, and moving logic across procedure boundaries.
- ★ **Idiomatic Variation:** Replacing language or library idioms with equivalent alternatives that preserve the scenario.

While the first three classes are semantics-preserving refactorings at different levels, **Idiomatic Variation** helps uncover semantically equivalent code patterns that realize the same scenario through alternative language or library idioms. The generated variants are then added back to the project and evaluated by the checker as additional test cases. All test cases are finally grouped into TP, FN, TN, and FP based on the classification rule in Section 2.2.

**Example:** Beyond the variants in Section 2.3, Figure 7 shows an idiomatic variation of *Pos#6* that exposes another false negative. Replacing `executeQuery()` with `execute()` and `getResultSet()` unexpectedly bypasses the checker and suppresses the alert.

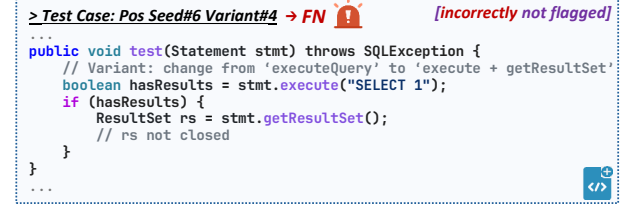


Figure 7: Another variant for *Pos#6* that demonstrates how Idiomatic Variation exposes a false negative.

## 4 Evaluation

We conduct extensive evaluations of SCENSCT to address the following targeted research questions:

- RQ1: (**Effectiveness**): Can SCENSCT generate test cases and independently identify FP/FN cases for static code checkers?
- RQ2: (**Generalizability**): Can SCENSCT generate tests across different tools and characterize capability differences within shared checkers?
- RQ3: (**Ablation Study**): What is the contribution of specific components and strategies of SCENSCT?
- RQ4: (**Cost**): What is the cost of using SCENSCT?

### 4.1 Experimental Setup

**4.1.1 Tool Selection.** Following prior work [32, 56], we select Java checkers from five widely studied static analysis tools: CodeQL, Checkstyle, Error Prone, PMD, and SonarQube. Table 1 summarizes the tools' basic statistics. These checkers' implementations frequently rely on shared framework utilities, meaning the direct source code may lack the full checking semantics. To improve semantic completeness, we implement an automated process to resolve and inline dependencies. We first parse the checker source code using Tree-sitter [10] to identify imported libraries and external API usages, then gather transitive dependencies within the directly imported files, limiting resolution depth to 1 to prevent context explosion. As shown in Table 1, this process increases the average lines of code (LoC) per checker by 4.0% (PMD) to 157.8% (CodeQL), which reflects the recovered checker logic.

**4.1.2 Dataset Construction.** To evaluate independent checker testing and cross-tool applicability, we construct two datasets from the five tools. Figure 8 presents the detailed construction workflows.

**Single-Tool Dataset I.** We use CodeQL for the independent checker-testing evaluation because it is actively maintained, widely adopted in both research and industry [16, 45], and covers diverse

Table 1: Statistics of the Selected Static Analysis Tools

| Tool            | Version      | # Checkers | Avg. Raw LoC / Checker | Avg. Resolved LoC / Checker |
|-----------------|--------------|------------|------------------------|-----------------------------|
| CodeQL [3]      | 2.24.3       | 451        | 21.8                   | 56.2                        |
| Checkstyle [2]  | 13.3.0       | 201        | 138.4                  | 160.9                       |
| PMD [5]         | 7.22.0       | 290        | 55.7                   | 57.9                        |
| SonarQube [6]   | 8.26.0.42915 | 626        | 97.0                   | 118.4                       |
| Error Prone [4] | 2.48.0       | 629        | 124.2                  | 142.9                       |

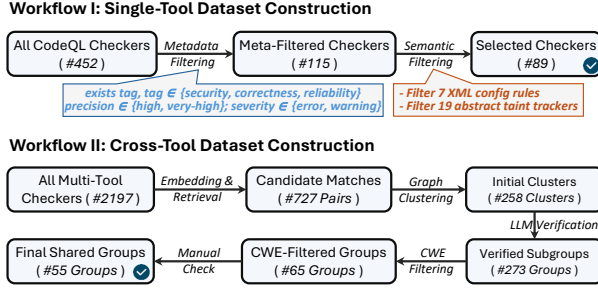


Figure 8: The workflows for constructing the two datasets.

static-analysis mechanisms such as AST-based matching and data-flow analysis [55]. As shown at the top of Figure 8, we focus on high-confidence defect-oriented checkers by retaining only the ones tagged with *security*, *correctness*, or *reliability*, with *high/very-high* precision and *error/warning* severity. Then, we exclude 7 XML configuration checkers and 19 template-based taint-tracking checkers whose specifications are hidden within deep abstractions. The final dataset contains **89** CodeQL checkers.

**Cross-Tool Dataset II.** To evaluate cross-tool applicability and characterize capability differences, we construct a focused dataset for differential testing [50], which consists of checker groups that target the same defect across different tools. Starting from all 2,197 checkers from the five tools, we first retrieve semantically similar checker pairs using rule-description embeddings (*Qwen3-Embedding* + FAISS [15], with a similarity threshold of 0.85). We then cluster these candidates with *NetworkX* [17] and use *DeepSeek-V3.2* [35] to verify that the checkers in each cluster target the same underlying defect. To prioritize security-related rules, we further filter out groups without any CWE tag, obtaining 65 groups, and then manually remove 10 groups that still lack strict rule equivalence (e.g., CodeQL’s *general* dead field vs. PMD’s only *private* dead field). The final dataset contains 55 shared-checker groups, comprising 50 checkers from CodeQL, 5 from Checkstyle, 18 from PMD, 38 from SonarQube, and 20 from Error Prone.

**4.1.3 Evaluation Settings.** To the best of our knowledge, no prior checker-testing method is directly comparable to SCENSCT in a head-to-head manner. In particular, STATFIER [56], the closest SOTA work, assumes existing manual seed test cases and focuses on test augmentation, whereas SCENSCT generates tests from scratch. Thus, we do not include a head-to-head baseline in RQ1 or RQ2. RQ1 evaluates independent FP/FN discovery, while RQ2 evaluates cross-tool applicability and capability differences. To assess the contribution of prior techniques, we include STATFIER-style augmentation in the ablation study (RQ3).

During evaluation, we manually assess the FP and FN candidates reported by SCENSCT for precision and uniqueness. Precision measures whether a candidate is a genuine FP/FN case according to the rule, while uniqueness captures whether it represents a distinct code pattern. Two developers, each with over three years of experience in Java static analysis, independently assess every candidate. Disagreements are resolved through discussion until consensus is reached, and uncertain cases are excluded.

**4.1.4 Implementation Details.** We implement SCENSCT using LangGraph [29] for multi-agent orchestration and Langfuse [30] for observability. DeepSeek-V3.2 serves as the sole LLM backend (temperature=0), selected for its demonstrated balance of code generation performance and inference cost and its wide adoption in recent work [35, 36, 54]. We use ANTLR v4 [42] for CheckerIR parsing, Tree-sitter [10] for source code analysis, and Gradle to build the generated Java test projects. We set maxDepth=3 and combStrategy=2-way for logic decomposition, limit Sub-IRs to 15 and scenarios to 10 per polarity, and allow up to 5 build attempts with one alignment round per test.

**Prompt Engineering.** We use three prompting techniques for LLM calls, following best practices from previous studies [33, 49, 58]: (1) *Specialized System Prompts* assign distinct expert roles for different subtasks based on their expertise. (2) *Unified Prompt Templates* follow a consistent ‘Goal-Rules-Task’ structure to ensure consistency, clarity, and cache-friendliness across all agents. (3) *Custom Chain-of-Thought (CoT)* integrates explicit task-oriented analysis steps directly into the output format as reasoning guidance.

## 4.2 RQ1: Effectiveness Evaluation

We evaluate test generation and independent FP/FN discovery on Dataset I. Table 2 shows the overall generation results. Specifically, SCENSCT extracted 765 checking scenarios and generated 3,501 compilable test cases, corresponding to 8.6 scenarios and 39.3 tests per checker on average. This result indicates that SCENSCT can expand one checker into multiple positive and negative scenarios and generate diverse test cases.

We next evaluate FP/FN exposure. As summarized in Table 3, SCENSCT flagged 360 raw FP/FN candidates across 71 of the 89 checkers. Manual assessment identified 311 genuine cases in 65 checkers, yielding 86.4% precision. After removing 41 duplicates, 270 unique FP/FN cases remained, including 58 FPs and 212 FNs. Of these, 100 cases (37.0%) were exposed directly by seed tests whose observed results still disagreed with their scenarios after refinement, while 170 cases (62.9%) were revealed after scenario-invariant augmentation, including 70 semantics-preserving refactorings and 100 idiomatic variations. Thus, both seed generation and augmentation contribute to FP/FN exposure. We also analyzed why FNs

Table 2: Overall generation results of SCENSCT on Dataset I.

| Metric                  | #Extracted Scenarios |     |     | #Generated Test Cases |       |       |      |
|-------------------------|----------------------|-----|-----|-----------------------|-------|-------|------|
|                         | pos                  | neg | all | pos                   | neg   | all   | seed |
| <b>Total Count</b>      | 291                  | 474 | 765 | 1,579                 | 1,922 | 3,501 | 754  |
| <b>Avg. per Checker</b> | 3.3                  | 5.3 | 8.6 | 17.7                  | 21.6  | 39.3  | 8.5  |

Table 3: FP/FN exposure results of SCENSCT on Dataset I.

| Type         | #Candidate FP/FN | #Assessed FP/FN | Precision    | #Unique Cases |
|--------------|------------------|-----------------|--------------|---------------|
| <b>FP</b>    | 80               | 65              | <b>81.3%</b> | 58            |
| <b>FN</b>    | 280              | 246             | <b>87.9%</b> | 212           |
| <b>Total</b> | <b>360</b>       | <b>311</b>      | <b>86.4%</b> | <b>270</b>    |

outnumber FPs and found that many checkers do not support certain language constructs or fully model their data flow, such as ternary expressions, aliases, and anonymous-class wrappers.

**Failure Analysis.** For the 49 discarded candidates, we further categorized them into three groups based on their root causes.

- *Scenario Extraction Misalignment (9, 18.4%)*: The extracted scenario is unrelated or misaligned with the rule, mainly due to LLM hallucinations during scenario extraction.
- *Test Generation Errors (33, 67.3%)*: The generated test case fails to preserve the intended semantics of the original scenario, mainly due to LLM hallucinations during test generation.
- *Test Execution Failures (7, 14.3%)*: The test follows the scenario, but the over-abstracted checker implementation makes the scenario too broad to yield a consistent result.

**Developer Feedback.** We reported 66 selected cases from 19 affected checkers to the CodeQL developers, and all of them were acknowledged as real defects. We only reported these 19 checkers because the CodeQL developers prioritize resolving issues with clearer real-world impact, and filing all remaining cases based on synthetic tests would impose a substantial triaging burden.

► **Answering RQ1:** SCENSCT can effectively extract multiple scenarios and generate diverse test cases for checkers. On 89 CodeQL checkers, it exposes 270 FP/FN defects with 86.4% precision.

### 4.3 RQ2: Generalizability Evaluation

We run SCENSCT on 55 shared-checker groups containing 131 checkers to evaluate cross-tool applicability and characterize capability differences. We first evaluate test generation across tools. As shown in Table 4, SCENSCT extracts multiple scenarios and generates tests for all five tools, with averages of 6–9 scenarios and 33–48 tests per checker. The same workflow also identifies 304 candidate FP/FN cases in the 81 non-CodeQL checkers. These results support its applicability across the five evaluated tools.

We then use differential testing to characterize the capability boundaries of shared checkers. A *differential FP/FN case* occurs when one checker misclassifies a test that at least one other checker handles correctly for the same shared rule. Under this definition, differential testing reveals how the practical detection capabilities of shared checkers vary across tools.

As shown in Table 4, SCENSCT exposes 177 differential cases in 63 of the 131 checkers, including 66 differential FPs and 111 differential FNs. These cases may arise because tools adopt different analysis mechanisms or rule-design strategies to balance precision and efficiency, and thus do not always represent actionable defects. For example, for *CWE-797: Unsafe Hostname Verification*, CodeQL reports a warning only when the unsafe verifier flows to a sink, whereas SonarQube flags the verifier directly through lightweight AST matching. Therefore, rather than viewing differential cases only as defects to be fixed, we use them as evidence of cross-tool capability differences. Such differences provide useful references for understanding the practical detection boundaries of shared checkers across tools.

**Table 4: Per-tool generation and differential-testing results of SCENSCT on Dataset II.**

| Tool           | Statistics per Checker |              | #Differential Cases |           |            |
|----------------|------------------------|--------------|---------------------|-----------|------------|
|                | Avg. # Scenarios       | Avg. # Tests | Affected Checkers   | Diff. FPs | Diff. FNs  |
| Checkstyle     | 7.00                   | 33.80        | 4 / 5               | 7         | 2          |
| CodeQL         | 7.72                   | 38.92        | 22 / 50             | 12        | 56         |
| Error Prone    | 8.20                   | 41.60        | 11 / 20             | 13        | 19         |
| PMD            | 6.56                   | 35.50        | 10 / 18             | 10        | 19         |
| SonarQube      | 8.89                   | 47.84        | 16 / 38             | 24        | 15         |
| <b>Overall</b> | <b>7.90</b>            | <b>41.30</b> | <b>63 / 131</b>     | <b>66</b> | <b>111</b> |

A *differential case* is an FP/FN case misclassified by one checker but correctly handled by at least one other checker for the same shared rule.

► **Answering RQ2:** SCENSCT generalizes well across different tools, with stable scenario-extraction and test-generation capabilities. It reveals 177 cross-tool capability differences on Dataset II.

### 4.4 RQ3: Ablation Study

The ablation study examines two design choices in SCENSCT: the logic-decomposition strategy in logic-based IR decomposition and the scenario-invariant test augmentation.

**Logic-Decomposition Strategy.** To sum up, Table 5 shows that `combStrategy=2-way` with `maxDepth=3` provides the best trade-off for logic decomposition. We evaluate the translated CheckerIRs of all 201 unique checkers in the union of Dataset I and Dataset II across all five tools, varying `maxDepth` ( $\in \{2, 3, 4\}$ ) and `combStrategy` ( $\in \{\text{all}, 2\text{-way}, 1\text{-way}\}$ ). 1-way is stable but expands too little. In contrast, all expands much more but can trigger an explosion, while 2-way improves expansion without causing an explosion. Under 2-way, both `maxDepth=3` and 4 generate enough candidate Sub-IRs for downstream filtering, since SCENSCT keeps at most 15 Sub-IRs per polarity. We adopt `maxDepth=3` as the default because manual inspection shows that many Sub-IRs from `maxDepth=4` are overly detailed and lead to trivial scenarios. Overall, 2-way with `maxDepth=3` best balances diversity and usefulness.

**Table 5: Hyperparameter selection for logic decomposition on the 201-checker union set across all five tools.**

| Hyperparameter |          | pos           | neg           |
|----------------|----------|---------------|---------------|
| combStrategy   | maxDepth | Avg. #Sub-IRs | Avg. #Sub-IRs |
| 1-way          | 2        | 2.2           | 5.9           |
| 1-way          | 3        | 2.9           | 7.0           |
| 1-way          | 4        | 3.4           | 7.8           |
| 2-way          | 2        | 2.7           | 6.6           |
| 2-way          | 3        | 4.7           | 8.9           |
| 2-way          | 4        | 8.5           | 12.1          |
| all            | 2        | 3.5           | 10.0 (1X)     |
| all            | 3        | 11.6 (1X)     | 41.4 (1X)     |
| all            | 4        | 36.6 (2X)     | 47.5 (6X)     |

X denotes decomposition explosion (reaching the 10 s timeout).

**Table 6: Comparison of augmentation strategies for SCENSCT on Dataset I.**

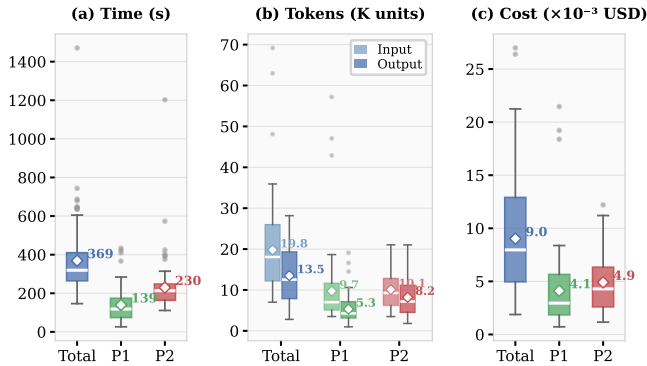
| SCENSCT Aug. Setting     | #Candidate<br>FP/FN | #Assessed<br>FP/FN | Precision | #Unique<br>Cases |
|--------------------------|---------------------|--------------------|-----------|------------------|
| w/o aug.                 | 112                 | 100                | 89.3%     | 100              |
| with STATFIER-style aug. | 176                 | 156                | 88.6%     | 156              |
| with SCENSCT aug.        | 360                 | 311                | 86.4%     | 270              |

**Scenario-Invariant Augmentation.** We further compare three augmentation settings: *w/o augmentation*, *STATFIER-style augmentation*, and the full SCENSCT. As shown in Table 6, the full method exposes 170.0% more unique FP/FN cases than no augmentation and 73.1% more than STATFIER-style augmentation. The gain over STATFIER-style augmentation shows the benefit of using LLMs to exercise more code forms under the same checking scenario. Moreover, unlike scenario-invariant augmentation, semantics-preserving transformations do not necessarily preserve the target checking scenario. In the STATFIER-style setting, we observe 8 variants that change the checking context and no longer instantiate the original scenario. For example, in `UseBraces.q1`, class wrapping introduces braces and changes the target checking context.

► **Answering RQ3:** For SCENSCT, logic decomposition with 2-way and `maxDepth=3` balances diversity and tractability, while Scenario-Invariant Augmentation exposes substantially more FP/FN cases than the other settings.

#### 4.5 RQ4: Cost Evaluation

As shown in Figure 9, the end-to-end cost of SCENSCT remains moderate. On average, processing one checker takes about 6 minutes, consumes **19.8K** input tokens and **13.5K** output tokens, and costs only  $9.0 \times 10^{-3}$  USD. This surprisingly low cost is due to the cache-friendly prompt design, deterministic workflow, and the cost-effective LLM backbone. The two major phases are also balanced, while test generation usually costs slightly more because of the build and augmentation processes. Although a few outliers incur higher time or token usage, the overall monetary cost stays low, remaining below **1 US cent per checker** on average.

**Figure 9: The detailed costs of SCENSCT per CodeQL checker.**

► **Answering RQ4:** SCENSCT is cost-efficient in practice. On average, processing one checker costs less than one US cent and takes around 6 minutes, making SCENSCT suitable for use at scale.

## 5 Threats to Validity

**Generalizability.** Our evaluation covers Java checkers from five tools, but does not include other target languages or unseen checker frameworks, which may limit the generalizability of our findings. To mitigate this threat, SCENSCT separates the core scenario-guided workflow from tool- and language-specific support. Both scenario extraction and test generation follow the same pipeline, while only checker runtime and project compilation require adaptation. RQ2 supports cross-tool applicability within the evaluated scope. For other languages, our implementation falls back to syntax validation, while compilation requires language-specific configuration.

**Practical Contribution.** Not every FP/FN case exposed by SCENSCT is equally likely to be fixed, because some behaviors reflect tool-specific trade-offs in precision or efficiency, or maintainer priorities. The practical role of SCENSCT is broader than bug reporting: it uses concrete tests to expose and document checker capability boundaries. RQ1 supports independent checker testing for developers, while RQ2 characterizes capability differences among tools for shared rules. The generated tests can also be retained for regression testing. Moreover, the practical value of some exposed cases is confirmed: all 66 selected cases reported to CodeQL developers were acknowledged as real defects. Thus, SCENSCT remains useful for understanding and validating checker behavior even when an exposed case is not prioritized for fixing.

**Reliance on LLM Components.** SCENSCT uses LLMs in multiple stages, including CheckerIR translation, scenario extraction and validation, seed generation and repair, and test augmentation. This architectural reliance may introduce hallucinations, propagate errors across stages, and reduce reproducibility. To mitigate this threat, we use LLMs for tasks that require interpreting heterogeneous checker implementations or generating API- and idiom-aware code, while placing deterministic checks between these stages. Grammar parsing, logic decomposition and conflict pruning, project validation, and checker execution constrain or check intermediate outputs. The rule description also anchors scenario validation, so checker behavior serves only as feedback rather than as the oracle. These measures cannot guarantee semantic correctness, but manual assessment yields 86.4% precision for the reported candidates, and RQ3 shows that the full augmentation exposes 73.1% more unique FP/FN cases than STATFIER-style transformations. Nevertheless, errors may still pass through these checks, and results may vary across models or runs even at temperature zero.

## 6 Related Work

### 6.1 Static Checker Testing

Prior work on static checker testing follows a black-box paradigm focused on two granularities. The first category treats the entire tool as a black box, identifying defects through dynamic execution feedback [14, 31, 51], static analysis features [19, 26], or differential testing [27, 57]. The second category treats individual checkers

as independent black boxes, generating test cases from rule descriptions and official examples [40, 56]. Specifically, Statfier [56] extracts seed programs from official test suites and documentation, and then applies semantics-preserving transformations to these seeds, detecting bugs when the checker’s output remains inconsistent between the original program and its variants.

SCENSCT advances the state-of-the-art by shifting from these documentation-reliant methods to an implementation-driven white-box paradigm. While existing tools rely on external artifacts like documentation or existing seed programs, our method extracts fine-grained scenarios directly from the checker source code. This approach ensures test cases rigorously verify the actual internal implementation logic, overcoming the scenario coverage gaps inherent in rule-based or documentation-driven generation.

## 6.2 LLM-Based Test Generation

LLM-based test generation has also been explored in unit, API, and compiler testing [21, 49]. Compiler testing shares the closest paradigm with static checker testing, as both require the synthesis of syntactically valid and semantically rich input programs. Existing LLM-based compiler testing mainly follows two directions. The first is fuzzing-oriented generation, which steers LLMs with richer semantic signals, such as reusable feature-rich code fragments, bug-derived semantic features, synthesized mutators, or compiler source code that reveals optimization-triggering conditions [20, 39, 41, 53]. The second is differential testing, which uses inconsistencies across implementations, versions, or compiler settings as the oracle, and it guides LLMs with specifications, code artifacts, or optimization cues to generate discriminative programs [22, 43].

While these methods have performed well in compiler testing, the compilers they target are extremely complex, making full coverage of their functional scenarios difficult. In contrast, this paper focuses on testing checkers and uses a white-box approach to extract scenarios from checker source code. It then generates corresponding compilable test snippets, enabling broader coverage of checker scenarios.

## 7 Conclusion

This paper presents SCENSCT, a white-box checker testing approach that synergizes the rule description and checker code to generate diverse test cases that instantiate rule-validated checking scenarios. To our knowledge, this is the first work that studies static checker testing from a white-box perspective and focuses on comprehensive testing at the checker level. Across 89 CodeQL checkers and 131 shared checkers from five tools, SCENSCT successfully generates diverse tests, exposes checker FP/FN issues, and reveals 177 cross-tool capability differences. Beyond bug finding, the generated tests can be reused as regression tests when checkers evolve. This makes SCENSCT useful as a practical support tool for checker development and maintenance. Future work includes extending SCENSCT to more tools and languages and adopting other advanced LLM backbones or techniques to expose more practical bugs.

## Acknowledgments

We thank all the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62132020.

## Data Availability Statement

The artifact supporting this paper, including the source code and data, is archived at <https://doi.org/10.5281/zenodo.19251689>. The maintained source code is available at <https://github.com/SQUARE-RG/ScenSCT>. An interactive visualization of the CodeQL-generated test cases and their manual annotations in Section 4.2 is available at <https://scensct-visualizer.pages.dev/>.

## References

- [1] 2025. The Top Programming Languages 2025. <https://spectrum.ieee.org/top-programming-languages-2025>.
- [2] 2026. Checkstyle. <https://checkstyle.org>.
- [3] 2026. CodeQL. <https://codeql.github.com>.
- [4] 2026. Error Prone. <https://errorprone.info/>.
- [5] 2026. PMD Source Code Analyzer. <https://pmd.github.io/>.
- [6] 2026. SonarQube. <https://www.sonarqube.org>.
- [7] Kus Andriadi, Haryono Soeparno, Ford Lumban Gaol, and Yulyani Arifin. 2023. The Impact of Shift-Left Testing to Software Quality in Agile Methodology: A Case Study. In *2023 International Conference on Information Management and Technology (ICIMTech)*. 259–264. doi:10.1109/ICIMTech59029.2023.10277919
- [8] Mordechai Ben-Ari. 2012. *First-Order Logic: Terms and Normal Forms*. Springer London, London, 167–183. doi:10.1007/978-1-4471-4129-7\_9
- [9] Steven Bird and Edward Loper. 2004. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics, Barcelona, Spain, 214–217. <https://aclanthology.org/P04-3031/>
- [10] Max Brunsfeld, Amaan Qureshi, Andrew Hlynyski, Will Lillis, ObserverOfTime, Christian Clason, et al. 2026. *tree-sitter/tree-sitter: v0.26.7*. doi:10.5281/zenodo.1902235
- [11] Hayden Cheers, Yuqing Lin, and Shamus P Smith. 2019. SPPlagiarise: A tool for generating simulated semantics-preserving plagiarism of Java source code. In *2019 IEEE 10th International conference on software engineering and service science (ICSESS)*. IEEE, 617–622. doi:10.1109/ICSESS47205.2019.9040853
- [12] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343. doi:10.1145/2970276.2970347
- [13] Han Cui, Menglei Xie, Ting Su, Chengyu Zhang, and Shin Hwei Tan. 2024. An Empirical Study of False Negatives and Positives of Static Code Analyzers From the Perspective of Historical Issues. arXiv:2408.13855 [cs.SE] doi:10.48550/arXiv.2408.13855
- [14] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods*, Alwyn E. Goodloe and Suzette Person (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 120–125. doi:10.1007/978-3-642-28891-3\_12
- [15] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2026. The Faiss Library. *IEEE Transactions on Big Data* 12, 2 (2026), 346–361. doi:10.1109/TBDATA.2025.3618474
- [16] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2025. Security weaknesses of Copilot-generated code in GitHub projects: An empirical study. *ACM Transactions on Software Engineering and Methodology* 34, 8 (2025), 1–34. doi:10.1145/3716848
- [17] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. 2008. Exploring network structure, dynamics, and function using NetworkX. Los Alamos National Laboratory (LANL). doi:10.25080/TCWV9851
- [18] John Harrison. 2009. *Handbook of Practical Logic and Automated Reasoning* (1st ed.). Cambridge University Press, USA. doi:10.1017/CBO9780511576430
- [19] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *Proc. ACM Softw. Eng.* 1, FSE, Article 74 (July 2024), 23 pages. doi:10.1145/3660781
- [20] Xingbang He, Yuanwei Chen, Hao Wu, Jikang Zhang, Zicheng Wang, Ligeng Chen, Junjie Peng, Haiyang Wei, Yi Qian, Tiantai Zhang, Linzhang Wang, and Bing Mao. 2026. Discovering 100+ Compiler Defects in 72 Hours via LLM-Driven

- Semantic Logic Recomposition. arXiv:2601.12360 [cs.SE] doi:10.48550/arXiv.2601.12360
- [21] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988
  - [22] Davide Italiano and Chris Cummins. 2025. Finding Missed Code Size Optimizations in Compilers using Large Language Models (CC '25). Association for Computing Machinery, New York, NY, USA, 81–91. doi:10.1145/3708493.3712686
  - [23] Adrián Herrera Jerónimo, Patricia Martínez Moreno, José Antonio Vergara Camacho, and Gerardo Contreras Vega. 2024. Techniques of SAST tools in the early stages of secure software development: A systematic literature review. In *2024 IEEE International Conference on Engineering Veracruz (ICEV)*. IEEE, 1–8. doi:10.1109/ICEV63254.2024.10766004
  - [24] Zongze Jiang, Ming Wen, Ge Wen, and Hai Jin. 2025. Fact-Aligned and Template-Constrained Static Analyzer Rule Enhancement with LLMs. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1642–1654. doi:10.1109/ASE63991.2025.00138
  - [25] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681. doi:10.1109/ICSE.2013.6606613
  - [26] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholtz, and Maria Christakis. 2024. Interrogation testing of program analyzers for soundness and precision issues. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 319–330. doi:10.1145/3691620.3695034
  - [27] Christian Klinger, Maria Christakis, and Valentin Wüstholtz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (Beijing, China) (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 239–250. doi:10.1145/3293882.3305053
  - [28] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421. doi:10.1109/TSE.2004.24
  - [29] LangChain, Inc. 2026. LangGraph. <https://github.com/langchain-ai/langgraph>. GitHub repository, accessed 2026-03-23.
  - [30] Langfuse. 2026. Langfuse. <https://github.com/langfuse/langfuse>. GitHub repository, accessed 2026-03-23.
  - [31] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An LLM-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499. doi:10.1145/3649828
  - [32] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and evaluation on static application security testing (SAST) tools for Java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 921–933. doi:10.1145/3611643.3616262
  - [33] Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W. Eric Wong. 2025. Evaluating large language models for software testing. *Comput. Stand. Interfaces* 93, C (April 2025), 13 pages. doi:10.1016/j.csi.2024.103942
  - [34] Yiyang Li, Hongqiu Wu, and Hai Zhao. 2022. Semantic-preserving adversarial code comprehension. In *Proceedings of the 29th International Conference on Computational Linguistics*. 3017–3028.
  - [35] Aixun Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-V3 technical report. arXiv preprint arXiv:2412.19437 (2024). doi:10.48550/arXiv.2412.19437
  - [36] Jun Liu, Yuanyuan Xie, Jiwei Yan, Jinhao Huang, Jun Yan, and Jian Zhang. 2025. Write Your Own CodeChecker: An Automated Test-Driven Checker Development Approach with LLMs. arXiv:2411.06796 [cs.SE] doi:10.48550/arXiv.2411.06796
  - [37] Panagiotis Louridas. 2006. Static code analysis. *Ieee Software* 23, 4 (2006), 58–61. doi:10.1109/MS.2006.114
  - [38] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 532–543. doi:10.1145/3533767.3534374
  - [39] Yunbo Ni and Shaohua Li. 2025. Interleaving large language models for compiler testing. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 815–841. doi:10.1145/3763079
  - [40] Elijah Nnorom, Md Basim Uddin Ahmed, Jiho Shin, Hung Viet Pham, and Song Wang. 2025. StaAgent: An Agentic Framework for Testing Static Analyzers. arXiv:2507.15892 [cs.SE] doi:10.48550/arXiv.2507.15892
  - [41] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. 298–312. doi:10.1145/3622781.3674171
  - [42] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.
  - [43] Nikitha Rao, Elizabeth Gilbert, Harrison Green, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, and Sarah Fakhoury. 2025. DiffSpec: Differential Testing with LLMs using Natural Language Specifications and Code Artifacts. arXiv:2410.04249 [cs.SE] doi:10.48550/arXiv.2410.04249
  - [44] Zachary P. Reynolds, Abhinandan B. Jayanth, Ugur Koc, Adam A. Porter, Rajeev R. Raj, and James H. Hill. 2017. Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. 55–61. doi:10.1109/SER-IP.2017..20
  - [45] Mingjie Shen, Akul Abhilash Pillai, Brian A. Yuan, James C. Davis, and Aravind Machiry. 2025. Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper). *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA048 (June 2025), 24 pages. doi:10.1145/3728923
  - [46] Tamás Szabó. 2023. Incrementalizing production CodeQL analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1716–1726. doi:10.1145/3611643.3613860
  - [47] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (San Diego, CA, USA) (CGO '20)*. Association for Computing Machinery, New York, NY, USA, 81–93. doi:10.1145/3368826.3377927
  - [48] Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT feedback: What experienced developers like about static analysis. In *Proceedings of the 26th Conference on Program Comprehension*. 64–73. doi:10.1145/3196321.3196327
  - [49] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* 50, 4 (April 2024), 911–936. doi:10.1109/TSE.2024.3368208
  - [50] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find bugs in static bug finders. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (Virtual Event) (ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 516–527. doi:10.1145/3524610.3527899
  - [51] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639121
  - [52] Jun Yan and Jian Zhang. 2006. Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 1. 385–394. doi:10.1109/COMPSAC.2006.33
  - [53] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735. doi:10.1145/3689736
  - [54] Chen Yang, Lin Yang, Ziqi Wang, Dong Wang, Jianyi Zhou, and Junjie Chen. 2025. Clarifying Semantics of In-Context Examples for Unit Test Generation. *CoRR abs/2510.01994* (2025). arXiv:2510.01994 doi:10.48550/ARXIV.2510.01994
  - [55] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience* 53, 7 (2023), 1472–1495. doi:10.1002/spe.3199
  - [56] Huaen Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 237–249. doi:10.1145/3611643.3616272
  - [57] Huaen Zhang, Yu Pei, Shuyun Liang, Zezhong Xing, and Shin Hwei Tan. 2024. Characterizing and Detecting Program Representation Faults of Static Analysis Frameworks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1772–1784. doi:10.1145/3650212.3680398
  - [58] Yining Zheng, Haiyang Wei, Jiahao Lu, Linqi Yin, Yunke Zhang, Chengguo Xu, Hetao Cui, Tianxiang Sun, Shuang Chen, and Xipeng Qiu. 2026. Investigating effective LLM-based in-context tool use: what matters and how to improve. *Front. Comput. Sci.* 20, 7, Article 2007323 (2026), 16 pages. doi:10.1007/s11704-025-41365-6