Do SAST Alerts Become Reproducible Tests? Evaluating Context-Enriched LLMs for Security Verification

Rubens Abraão da Silva Sousa
State University of Ceará
Fortaleza, Ceará, Brazil
abraao.sousa@aluno.uece.br

Gabriel Pinheiro Rezende de Lima
State University of Ceará
Fortaleza, Ceará, Brazil
gabriel.rezende@aluno.uece.br

Samuel Sousa Teles
State University of Ceará
Fortaleza, Ceará, Brazil
samuel.teles@aluno.uece.br

Sávio Freire
Federal Institute of Ceará
Morada Nova, Ceará, Brazil
State University of Ceará
Fortaleza, Ceará, Brazil
savio.freire@uece.br

Ismayle de Sousa Santos
State University of Ceará
Fortaleza, Ceará, Brazil
ismayle.santos@uece.br

Abstract

Context. Static application security testing (SAST) tools report potential vulnerabilities, but their alerts rarely provide enough guidance for practical verification. **Goal**. This study investigates whether common weakness enumeration (CWE) descriptions, web security testing guide (WSTG) procedures, and local code snippets improve LLM-generated security verification procedures from SAST alerts. **Method**. We conducted a controlled repeated-measures experiment with eight SonarQube issues from OWASP Juice Shop. For each issue, procedures were generated under three conditions: isolated SAST alert, alert enriched with CWE/WSTG via retrieval-augmented generation (RAG), and alert enriched with CWE/WSTG plus local code. The procedures were evaluated by judges experienced in software testing, SAST, and pentesting. The rubric measured technical correctness, practical reproducibility, structural completeness, procedural clarity, WSTG alignment, and manual adjustments. **Results**. Contextual enrichment did not produce statistically consistent improvements in technical correctness, practical reproducibility, procedural clarity, or WSTG alignment. Practical reproducibility remained low across all conditions. The condition with local code showed the lowest median number of manual adjustments, but the effect did not remain significant after correction. **Conclusion**. Adding security documentation and local code to LLM prompts was not sufficient to transform SAST alerts into reproducible verification procedures. LLMs may support initial drafting, but reliable operationalization still requires human review, execution-based validation, and richer runtime context.

Keywords

Static Application Security Testing (SAST), Software Security, Vulnerability Analysis, LLMs

1 Introduction

Static Application Security Testing (SAST) tools are integrated into continuous integration and continuous delivery (CI/CD) pipelines and operate as the first phase of vulnerability identification in source code throughout the development lifecycle. Ami et al. [1] interviewed 20 professionals and identified that the absence of alert messages with practical meaning and a considerable effort required for configuration were reported as negative aspects of SAST tools. Furthermore, Lipp et al. [9] demonstrated that SAST tools for the C programming language, when analyzing 27 projects with 192 previously known vulnerabilities, failed to detect 47% to 80% of the cases. In these two scenarios, the high noise in the output and the incomplete coverage shift the security cost from the scanner to the professionals, who must triage each report, verifying its legitimacy.

The verification of a SAST alert requires a translation that the tool does not perform on its own. The evaluator must convert a static description into an exploitable hypothesis, relate the reported rule to a weakness category, identify relevant inputs or execution flows, and formulate steps compatible with the application's environment. This work combines security knowledge, code comprehension, and procedural domain on how to test the vulnerability at runtime.

The alerts provided by SAST tools, by their nature, point to potential vulnerabilities in the source code, but they do not deliver the tests and concrete guidance that confirm their exploitability. To carry out the verification of an alert, technical knowledge is required on three fronts: i) attack vector, scenario reproduction path, and vulnerability category; ii) procedure, understanding verification frameworks such as the OWASP Web Security Testing Guide (WSTG); and iii) translation, understanding the elements in executable steps that exercise the flagged vulnerable code.

Performing the verification of a SAST alert is an activity that becomes burdensome even for intermediate knowledge profiles. The initial barrier focuses on the conceptual understanding of the vulnerability, navigating toolchains, and coordinating verification steps to confirm exploitability [13]. Given this barrier, the SAST report remains underutilized because, even if it presents the problem, the tool does not deliver an instrument that assists in active verification.

Given the challenges presented by SAST tools, Large Language Models (LLMs) emerge as a possibility to assist the static verification of code vulnerabilities. Thus, it was observed that, when dealing with code context at the function level, LLMs can achieve a success rate of 9% to 13%. However, when observing the context at the code file level, their success rate is almost null [4, 20]. Studies by

Çetin and Bıyıklı [21] and Du et al. [3] demonstrate that LLMs generate vulnerable code recognized by the OWASP Top 10 [11]. However, studies show that LLMs have a promising capability to identify the vulnerability at its root cause, allowing the search for solutions that assist professionals in performing formal verification based on operationalization and verification frameworks, such as the OWASP WSTG.

Despite recent advances in combining SAST tools and LLMs for vulnerability detection and classification, the literature remains concentrated on the automatic identification of flaws or the generation of code fixes. Little is known about the capability of these models to transform SAST alerts into reproducible verification procedures that can be effectively executed by professionals during security verification activities. In particular, to the best of our knowledge, no studies were identified that experimentally evaluate the impact of contextual enrichment with structured security knowledge, such as the Common Weakness Enumeration (CWE), OWASP WSTG, and local code snippets, on the operational quality of test procedures generated by LLMs.

This study seeks to understand how the use of LLMs with the Retrieval-Augmented Generation (RAG) technique can assist in the documentation of security test cases based on the WSTG guidelines, leading to the following research question (RQ): **to what extent do CWE/WSTG knowledge and local code influence LLM-generated security verification procedures from SAST alerts?**

To answer the RQ, this work is conducted through a controlled repeated measures experiment [18] with eight issues reported by SonarQube[1] in the OWASP Juice Shop. For each issue, we generated procedures under three conditions: isolated SAST alert; alert augmented with CWE/WSTG documents retrieved via RAG; and alert augmented with CWE/WSTG and a local code snippet. The procedures were evaluated by judges with experience in testing, SAST, and pentesting, using a rubric that measured technical correctness, practical reproducibility, structural completeness, procedural clarity, alignment with WSTG, and manual adjustments.

The results did not indicate a statistically consistent improvement of the enriched conditions in technical correctness, practical reproducibility, procedural clarity, or alignment with WSTG. Practical reproducibility remained low across all three conditions, suggesting that the retrieval of CWE/WSTG knowledge and the inclusion of local code are not sufficient to generate executable procedures from SAST alerts. The condition with local code presented a lower median of manual adjustments, but this signal did not remain significant after correction for multiple comparisons.

Consequently, the present work seeks to contribute to the literature from the perspective of using a specific RAG architecture for the generation of security test cases from SAST reports addressed with exploit practices, providing professionals and researchers with an initial baseline to evaluate the feasibility of LLM-assisted generation in CI/CD-integrated SAST workflows.

This work is organized as follows: Section 2 presents the related work on three fronts: SAST and the alert operationalization problem, LLMs in security tasks, and RAG applied to software engineering. Section 3 presents the research method, including the RAG-based retrieval pipeline, prompt composition, experimental design, evaluation procedure, and statistical analysis. Section 4 reports the results by hypothesis, presenting the statistical tests. Section 5 discusses the findings and their practical implications. Section 6 presents the threats to validity and the precautions taken throughout the study. Finally, Section 7 concludes the paper, indicating future research directions.

2 Related Work

Security assurance in the software development life cycle has traditionally relied on SAST tools for the early detection of vulnerabilities. However, as reported by Ami et al. [1], the excess of false alarms and alert messages devoid of exploratory context represent a high cognitive barrier, overloading developers and testers. This gap between alert identification and its practical validation is the main motivation for seeking automated triage approaches.

To circumvent the limitations of SAST in the semantic identification of code, LLMs are being integrated into application security toolchains. Tihanyi et al. [15] evaluated the capability of eight LLMs in detecting vulnerabilities mapped in the Common Vulnerabilities and Exposures (CVE). The authors showed that LLMs can identify flaws that static models ignore. Despite this, the authors pointed out that no model achieved satisfactory efficacy autonomously.

Li et al. [7] conducted a comparative study of SAST tools for Java, in which SonarQube, Checkmarx, and Bandit were evaluated using TPR, FPR, precision, recall, and F1-score metrics against industry benchmarks, including the OWASP Top 10. Through the study, the authors demonstrated that modern SAST tools achieve variable detection rates depending on the vulnerability category. These tools identify vulnerabilities with measurable accuracy, but the generated alerts typically consist of technical descriptions that provide little operational guidance for verifying whether the reported vulnerability actually exists.

Aiming to leverage the strengths of each technology, hybrid approaches between static analysis tools and generative artificial intelligence have been proposed. [12] showed that the combination of traditional static analyzers with LLMs mitigates the rigidity of syntactic rules and broadens the detection of complex vulnerabilities. In line with this perspective, Li et al. [8] presented the neuro-symbolic framework IRIS, which utilizes the SAST architecture to guide the LLM's context, increasing the hit rate and minimizing false discoveries, evidencing the potential of collaboration between structured static rules and semantic reasoning. This line of research consolidates the premise that the collaboration between structured static rules and the semantic reasoning of language models is the most promising path for software security.

Despite the success in combined detection, the transition from the mere identification of flaws to the generation of practical artifacts still presents substantial challenges. Without the support of rigorous frameworks or documentation, LLMs tend to generate hallucinations and produce test plans that do not reflect the technical reality of the application [3]. The models cannot, relying solely on their own pre-trained knowledge, consistently follow official security standards, resulting in verbose responses or inexecutable steps. The literature establishes, therefore, that delegating the creation of

[1] https://www.sonarsource.com/products/sonarqube/

operational security routines to the LLM requires the continuous injection of specific technical knowledge during inference.

Wang et al. [16] presented a systematic survey and a comprehensive review regarding software testing with LLMs, reporting that LLMs have proven efficient during the intermediate and late phases of the testing life cycle, including the generation of unit test cases. The review identified an emerging pattern, which consists of LLMs demonstrating strong performance in test code generation tasks when provided with the appropriate context. However, a large part of the mapped studies evaluates the technical quality of the generated tests, leaving a gap regarding the actionability of the artifacts for novice testers, as there are no metrics to measure whether an inexperienced professional can interpret the generated tests.

To correct this context deficiency and anchor the model's reasoning, RAG technique has been consolidated in software engineering [6]. This approach enriches the input prompt with relevant documentation retrieved from vector databases, binding the artificial intelligence's response to official guidelines. In this sense, Shi and Zhang [14] proved the efficacy of this strategy with the RESCUE framework, demonstrating that by providing the LLM with explicit security guidelines and dynamically retrieved code snippets, the quality and security of the artifacts increased drastically. RAG acts as an essential mediator, preventing the AI from basing its results on generic statistical approximations and forcing it to use documented and validated logic.

However, while the use of SAST guided by LLMs and RAG strategies demonstrates maturity for code self-correction and reduction of false positives, there is an evident gap in the operationalization of reproducible verification tests. Previous works [8, 12, 15] focused on having the artificial intelligence scan the code or rewrite it, but ignored the analyst's daily need to transform a noisy alert into an efficient test case. The present study seeks to use RAG to cross-reference the static report of SAST tools, such as SonarQube, with official methodologies (CWE, OWASP WSTG) and local source code snippets.

In summary, the literature demonstrates consistent advances in the use of LLMs for vulnerability detection, classification, and correction, as well as in the use of RAG for the contextual enrichment of security tasks. However, the problem of operationalizing SAST alerts into reproducible and actionable verification procedures by professionals remains unexplored. Although previous works have investigated the capability of models to identify or correct vulnerabilities, no studies were found that empirically evaluate how different sources of security context influence the generation of executable validation procedures. This work seeks to fill this gap through a controlled experimental evaluation involving SAST alerts, structured security knowledge, and procedures derived from the OWASP WSTG.

3 Research Method

This section presents the phases of the experimental method used to evaluate the capability of LLMs in operationalizing SAST alerts as reproducible vulnerability verification procedures. The study compared three conditions of contextual enrichment: isolated SAST alert, SAST alert with CWE/WSTG knowledge retrieved via RAG, and SAST alert with CWE/WSTG augmented by a local code snippet.

The unit of analysis was the procedure generated for each combination of issue and experimental scenario. The evaluation combined an ordinal rubric, practical execution by judges, inter-rater agreement analysis, and statistical comparison across scenarios. The section is organized into three parts: 3.1 Experimental Design; 3.2 Experiment Procedures; and 3.3 Experiment Analysis.

3.1 Experimental Design

The experiment was planned according to the software engineering experimentation guidelines proposed by Wohlin et al. [18], observing how contextual enrichment with LLMs using security artifacts enables the operationalization of SonarQube SAST alerts. Thus, our experiment presents the following aims:

> ***Analyze*** *security verification procedures generated by LLMs for the* ***purpose*** *of evaluating their technical quality and capability to support the reproduction of vulnerabilities reported by a SAST tool* ***from the perspective*** *of researchers and professionals in software testing, SAST, and pentesting, in the* ***context*** *of vulnerabilities mapped from the OWASP Juice Shop.*

The null hypothesis (H_0) states that there is no difference among the three experimental conditions regarding the quality of the generated procedures. In contrast, the first alternative hypothesis (H_1) states that the use of CWE and WSTG improves technical correctness, procedural clarity, and alignment with the corresponding WSTG procedure when compared to the isolated use of the SAST alert. The second alternative hypothesis (H_2) states that the inclusion of the local code snippet reduces the number of manual adjustments and improves the practical reproducibility of the generated procedures when compared to the other conditions.

Based on the hypotheses, the study separates the independent variables by the level of contextual enrichment provided to the model, established in three scenarios (S): (S1) the prompt received only the SAST alert metadata, including rule, severity, file, flagged line, and SonarQube message; (S2) the prompt received the same metadata augmented with CWE descriptions and WSTG procedures retrieved via RAG; and (S3) the prompt received the elements of S2 and the local code snippet associated with the flagged line. The local snippet corresponded to the complete function when the line was contained within an identifiable function; in other cases, the smallest logical block that preserved the flagged instruction and its immediate dependencies was used.

The RAG pipeline was implemented using a persistent ChromaDB as the vector database and embeddings generated by the text-embedding-3-small model. Two document collections were used: CWE docs, containing CWE descriptions, and WSTG docs, containing documents from the OWASP WSTG. The collections were configured with cosine distance, filtered via metadata. The documents were segmented into chunks of 6000 characters, with an overlap of 400 characters. For each finding, the pipeline retrieved the two CWE documents and the two WSTG documents that were

semantically closest, totaling a maximum of four retrieved documents per generation.

The dependent variables were evaluated using a structured rubric. Technical correctness, practical reproducibility, and structural completeness were measured on a three-level ordinal scale. Procedural clarity was measured on a five-level ordinal scale. Alignment with WSTG was treated as a binary variable, indicating whether or not the procedure followed the corresponding WSTG procedure. Manual adjustments were recorded as a count variable, representing the number of modifications, additions, or reinterpretations made by the evaluator to conclude the execution. Table 1 presents the definitions of the evaluated variables.

To reduce variations resulting from the textual format of the responses, all prompts used a fixed output structure. This structure required the model to generate verification procedures containing, at a minimum: identification: SonarQube ID, File, Line, SAST Severity, Rule, Category; classification: Verdict, CWE, OWASP Top 10, WSTG; Justification, Technical Description, Preconditions for verification, Verification steps, Expected result (true positive), and Expected result (false positive). The only difference among the conditions was the context provided to the model prior to generation.

Prior to the formal evaluation, the judges participated in a synchronization phase using examples external to the experimental corpus. This phase aimed to align the interpretation of the rubric criteria and reduce subjective discrepancies during the evaluation of the generated procedures.

prompt used for generating verification procedures

Role instruction. You are a senior AppSec/SAST analyst.
Task instruction. Your task is to analyze a SonarQube alert and create a test case for verification and triage based on the fields below.
Output constraint. Return valid JSON only.
Required JSON structure.

```
{
  "test_cases": [
    {
      "identification": {
        "sonarqube_id": "...",
        "file": "...",
        "line": 0,
        "sast_severity": "...",
        "rule": "...",
        "category": "..."
      },
      "classification": {
        "verdict": "...",
        "cwe": "...",
        "owasp_top_10": "...",
        "wstg": "...",
        "justification": "..."
      },
      "technical_description": "...",
      "verification_preconditions": ["..."],
      "verification_steps": [
        {
          "step": "Step 01",
          "expected_result_true_positive": "...",
          "expected_result_false_positive": "..."
        }
      ]
    }
  ]
}
```

Generation constraint. Do not invent findings. Generate one test case per finding, in the same order.

The variables, in turn, follow these definitions:

- Technical correctness: the set of steps and the completion of the generated fields correctly represent the vulnerability mechanism, the exploitation vector, and the expected result.
- Practical reproducibility: the judges can execute the procedure and observe the expected behavior of the vulnerability by following exclusively the generated procedure.
- Structural completeness: presents a logical sequence, lacks ambiguity, and provides sufficient instructions for execution.
- WSTG alignment: follows the set of best practices for conducting penetration tests; Manual adjustments: corrections made by the evaluators to meet the expected standard.

3.2 Experiment Procedures

Initially, each issue in the corpus was converted into a standardized input containing information from the SAST alert, including rule, severity, associated CWE, and the line flagged by SonarQube. Subsequently, the experimental pipeline generated the procedures using three distinct experimental conditions. The generated outputs were anonymized and randomized prior to human evaluation to reduce judgment bias.

After the generation of the procedures, each judge executed the produced steps in an isolated environment, using Docker containers with version v20.0.0 of the OWASP Juice Shop[2] and recording each execution. After each execution, the environment was restarted to prevent interference between tests. Following each execution, the judges filled out a structured evaluation document, recording the application of the rubric, the difficulties encountered, the adaptations made, and observations regarding operationalization failures of the generated procedure.

The experimental corpus was composed of eight issues reported by SonarQube in OWASP Juice Shop v20.0.0. The issues covered vulnerabilities associated with XSS, code injection, credential exposure, cryptographic failures, permissive CORS configuration, and ReDoS. The selection sought to include alerts with the possibility of practical verification within the application environment. Each issue was associated with a SAST rule, a CWE, and a reference WSTG procedure.

To improve transparency about the generated artifacts, Table 2 presents an illustrative example from the same SonarQube finding across the three experimental scenarios. The example refers to the alert `typescript:S1523` in `routes/b2bOrder.ts`, associated with dynamic code execution. The table does not constitute a qualitative analysis; it illustrates how the amount of contextual information changed the generated verification procedure and how this variation could affect evaluator judgment. The example shows that contextual enrichment changed the specificity of the generated procedure. S1 produced a generic verification plan centered on the existence of dynamic execution and possible external influence. S2 added security-oriented terminology and checks derived from CWE/WSTG. S3 incorporated application-specific elements from

[2]https://github.com/juice-shop/juice-shop/releases/tag/v20.0.0

Table 1: Evaluation rubric for the generated procedures

Variable	Item	Definition
Technical correctness	0 – Incorrect	The procedure exploits a different mechanism, incorrectly interprets the alert, or produces an incompatible result.
	1 – Partially correct	The procedure partially identifies the correct vector but presents technical errors, omissions, or inadequate steps.
	2 – Correct	The procedure correctly identifies the vector, describes consistent steps, and predicts the expected behavior.
Practical reproducibility	0 – Does not reproduce	The procedure does not execute or does not demonstrate the expected behavior.
	1 – Partially reproduces	The procedure executes after adaptations or reproduces only part of the expected behavior.
	2 – Completely reproduces	The procedure executes as described and completely demonstrates the expected behavior.
Structural completeness	0 – Does not cover	Does not present the expected structure.
	1 – Partially covers	Presents the structure with some missing fields.
	2 – Covers	Presents all mandatory fields filled.
Procedural clarity	1 – Very low	The procedure is ambiguous or insufficient for execution.
	2 – Low	The procedure presents a general intent but has relevant gaps that hinder execution.
	3 – Moderate	The procedure is understandable and partially executable but requires complementary interpretation.
	4 – High	The procedure is clear and ordered, requiring only minor decisions from the evaluator.
	5 – Very high	The procedure is immediately understandable and executable without relevant additional interpretation.
WSTG alignment	Correct	The procedure is aligned with the corresponding WSTG procedure.
	Incorrect	The procedure is not aligned with the corresponding WSTG procedure.
Manual adjustments	Quantitative	Number of modifications made by the judge to complete the execution of the procedure.

the local code, including `body.orderLines Data`, `safeEval`, and `vm.runInContext`. However, even the most specific scenario still required runtime assumptions about route reachability, challenge configuration, and observable behavior.

The judges responsible for the evaluation include a QA specialist with three years of experience in software testing, pentesting, and application SAST reports, and a software testing specialist with two years of experience, including one year with SAST. In situations of divergence, a senior software testing researcher conducted the final arbitration.

To illustrate the judges' evaluation, Table 3 presents one example of evaluator judgment for the same generated procedure. The example is not used as qualitative evidence; it only illustrates how practical reproducibility and manual adjustments may diverge even when evaluators agree on most rubric dimensions. This example shows why practical reproducibility was harder to assess than text-oriented criteria. The evaluators agreed that the procedure was partially correct, partially complete, unclear, and not aligned with WSTG. However, they differed on whether the artifact could be partially reproduced and on how many manual adjustments were needed. This reinforces the need to treat practical reproducibility as a stricter criterion than structural completeness or technical plausibility.

As a reference for evaluating the generated procedures, a golden set[3] was constructed, containing verification procedures for each vulnerability in the experimental corpus. For each issue, a verification procedure was manually elaborated containing preconditions, execution steps, and expected results. The construction considered the SAST alert, the associated CWE documentation, the corresponding WSTG procedure, and the observable behavior of the vulnerability in the OWASP Juice Shop.

[3]https://github.com/ASTRID-RESEARCH/gold-set-sast-for-llm/

The golden set was made available to the judges during the evaluation. Thus, the evaluation should be interpreted as a reference-assisted evaluation, rather than a completely independent execution based solely on the generated procedure. The judges used the golden set to support judgments regarding technical correctness, alignment with WSTG, and the expected result of the verification. Experimental blindness was preserved regarding the procedure generation condition and the identification of the artifacts, but not regarding the expected reference for each vulnerability.

The material was reviewed by one security specialist with six years of experience and one QA with pentesting experience and four years of experience, prior to its use in the experiment. The review verified whether each procedure correctly described the vulnerability mechanism, whether the steps were executable in the experimental environment, and whether the expected result allowed distinguishing a true positive from a false positive. During the evaluation, the golden set was used as a reference to support judgments regarding technical correctness, WSTG alignment, and practical reproducibility.

3.3 Experiment Analysis

The experiment utilizes a repeated measures design, in which the same eight issues were evaluated under the three experimental conditions. For each combination of issue and experimental condition, a single final response generated by the LLM was considered, maintaining the model, temperature, generation parameters, and prompt structure fixed. Thus, the unit of quantitative analysis corresponded to the procedure generated for each issue-condition combination, totaling 24 evaluated procedures. This decision was adopted to keep the experimental design simple and controlled, reducing the influence of subsequent choices among multiple generated responses.

The evaluation of the procedures was conducted blindly. The judges did not have access to the experimental condition responsible for generating the procedure, nor to the identification of the model used. The artifacts were presented using randomized identifiers to

Table 2: Illustrative example of generated artifacts across scenarios for the same SAST finding

Scenario	Context provided	Generated classification	Verification focus	Potential evaluator concern
S1	SAST alert only	Classified as a security hotspot to be verified; mapped to CWE-95 and OWASP A03 Injection.	Focuses on confirming whether dynamic execution exists and whether external input reaches the flagged line.	The procedure remains generic and depends on the evaluator to identify the concrete endpoint, input field, and execution path.
S2	SAST alert + CWE/WSTG via RAG	Classified as code injection / eval injection; mapped to CWE-95/CWE-94 and WSTG code injection testing.	Adds more explicit checks for dynamic execution, data flow, input validation, and non-destructive payloads.	The procedure provides richer security framing, but still requires the evaluator to infer the concrete route and payload placement.
S3	SAST alert + CWE/WSTG + local code	Classified as conditional true positive; identifies `body.orderLinesData`, `safeEval`, and `vm.runInContext`.	Targets the concrete data flow from request body to dynamic execution and proposes traceable payloads such as `SAST_RCE_PROBE`.	The procedure is more application-specific, but still depends on challenge configuration, route reachability, and runtime behavior.

Table 3: Illustrative rubric application for one generated procedure

Criterion	Evaluator A	Evaluator B
Technical correctness	1 – Partially correct	1 – Partially correct
Practical reproducibility	1 – Partially reproduces	0 – Does not reproduce
Structural completeness	1 – Partially covers	1 – Partially covers
Procedural clarity	1 – Very low	1 – Very low
WSTG alignment	Incorrect	Incorrect
Manual adjustments	3	7

Note. The main disagreement occurred in practical reproducibility and manual adjustments. This example illustrates how evaluators may agree on textual criteria while differing on how much reconstruction is needed for execution.

minimize biases related to the size, complexity, or textual formatting of the generated outputs.

The data analysis was conducted using non-parametric statistical techniques due to the small size of the experimental corpus and the ordinal nature of the evaluated variables. Initially, we calculated the agreement between the two evaluators. For the ordinal variables of the rubric, the weighted Gwet's AC2 coefficient was used, accompanied by the observed agreement percentage and Cohen's weighted kappa as a complementary measure. For the binary variable of alignment with the WSTG, Gwet's AC1 coefficient was used, also accompanied by the observed agreement percentage and simple Cohen's kappa. For the quantitative variable of manual adjustments, we described the agreement through the mean and median absolute difference between evaluators, with the intraclass correlation coefficient reported as a complementary analysis.

After the agreement analysis, divergences between evaluators was resolved by consensus, producing a final score for each issue-condition combination. For the ordinal variables, we used the Friedman test for a global comparison among the three experimental conditions. When a statistically significant difference is identified, we performed paired comparisons using the Wilcoxon signed-rank test, with Holm's correction for multiple comparisons. The global effect size was reported using Kendall's W coefficient, associated with the Friedman test, while the paired comparisons was accompanied by the rank-biserial correlation. For the binary variable of alignment with the WSTG, we used Cochran's Q test as a global comparison, followed by paired McNemar's tests with Holm's correction when applicable. The variable of manual adjustments, treated as a count, was described by the median and interquartile range and compared among conditions using the Friedman test, followed by the paired Wilcoxon test when necessary.

The inter-rater agreement was analyzed using complementary metrics, considering the skewed distribution of the rubric responses. For the ordinal variables, we used Gwet's AC2 coefficient, while for the binary variable of alignment with the WSTG, we used Gwet's AC1. The observed agreement percentage and Cohen's kappa were reported as complementary measures.

4 Results

This section presents the results obtained in the experiment.

4.1 Agreement Among the Evaluators

In general, the results indicated high agreement for technical correctness (AC2 = 0.858; observed agreement = 87.5%), structural completeness (AC2 = 0.909; observed agreement = 91.7%), and procedural clarity (AC2 = 0.950; observed agreement = 95.7%). Alignment with the WSTG also presented substantial agreement (AC1 = 0.744; observed agreement = 79.2%). In contrast, practical reproducibility presented low inter-rater agreement (AC2 = 0.242; observed agreement = 54.2%), suggesting greater difficulty in applying this criterion.

Although the kappa coefficients were low or null for some variables, these values were interpreted with caution due to the heavy concentration of responses in predominant categories. Thus, the AC1/AC2 coefficients and the observed agreement were considered more informative for evaluating the consistency among evaluators in this study.

4.2 Descriptive Statistics by Scenario

Table 4 presents the descriptive statistics of the variables evaluated across the three experimental scenarios, S1, S2, and S3. For each variable, the number of observations N, the median, the quartiles Q1 and Q3, as well as the minimum and maximum values are reported.

In general, the variables technical correctness, structural completeness, and procedural clarity presented a median equal to 1 across the three experimental scenarios. In contrast, practical reproducibility presented a median equal to 0 in all scenarios. The manual adjustments variable was the only one that presented more evident differences among the scenarios, with medians of 7.0 in S1, 8.5 in S2, and 6.0 in S3.

Although the median for technical correctness was equal to 1 in all three scenarios, the occurrence of minimum values equal to 0 was observed in S1 and S2, indicating that not all scenarios reached the highest level of correctness. In scenario S3, all scenarios presented a minimum value equal to 1.

Practical reproducibility presented a median equal to 0 in all experimental scenarios. Furthermore, S1 and S2 did not present

Table 4: Descriptive statistics by scenario.

Variable	Scenario	N	Median	Q1	Q3	Min	Max
Technical correctness	S1	8	1,00	0,75	1,00	0,00	2,00
Technical correctness	S2	8	1,00	1,00	1,00	0,00	2,00
Technical correctness	S3	8	1,00	1,00	1,00	1,00	2,00
Practical reproducibility	S1	8	0,00	0,00	0,00	0,00	2,00
Practical reproducibility	S2	8	0,00	0,00	0,00	0,00	2,00
Practical reproducibility	S3	8	0,00	0,00	0,00	0,00	2,00
Structural completeness	S1	8	1,00	1,00	1,00	1,00	1,00
Structural completeness	S2	8	1,00	1,00	1,00	1,00	1,00
Structural completeness	S3	8	1,00	1,00	1,00	1,00	1,00
Procedural clarity	S1	8	1,00	1,00	1,00	1,00	2,00
Procedural clarity	S2	8	1,00	1,00	1,00	1,00	2,00
Procedural clarity	S3	8	1,00	1,00	1,00	1,00	2,00
Manual adjustments	S1	8	7,00	5,00	8,00	5,00	10,00
Manual adjustments	S2	8	8,50	6,75	13,00	6,00	18,00
Manual adjustments	S3	8	6,00	6,00	7,00	5,00	7,00

variation among the evaluated scenarios, while S3 presented a single case with a maximum value equal to 1, indicating minor variation in this scenario.

Regarding manual adjustments, scenario S2 presented the highest median (8.5) and the greatest dispersion of values, ranging from 6 to 18 adjustments. Scenario S1 presented a median of 7.0, while S3 presented the lowest median (6.0) and the lowest variability among the evaluated scenarios, with values between 5 and 7 adjustments.

Although the enriched scenarios provided more context to the model, this enrichment was not reflected in more easily executable procedures, as evidenced by the low reproducibility observed across the three scenarios.

4.3 Global Comparison Between Scenarios

Table 5 presents the results of the global statistical tests used to compare the three experimental scenarios S1, S2, and S3. For each evaluated variable, the applied statistical test, the test statistic, the degrees of freedom (df), the p-value, the number of observations N, and, when applicable, Kendall's coefficient of concordance (W), used as an effect size measure for the Friedman test, are reported.

Table 5: Global tests comparing scenarios.

Variable	Test	Statistic	gl	p	N	Kendall_W
Technical correctness	Friedman	3,000	2	0,223	8	0,188
Practical reproducibility	Friedman	2,000	2	0,368	8	0,125
Structural completeness	Friedman	–	2	–	8	–
Procedural clarity	Friedman	2,000	2	0,368	8	0,125
Manual adjustments	Friedman	6,690	2	0,035	8	0,418
Alignment WSTG	Cochran Q	2,000	2	0,368	8	–

The global analyses did not identify statistically significant differences among scenarios S1, S2, and S3 for the variables Technical correctness (p=0.223), Practical reproducibility (p=0.368), Procedural clarity (p=0.368), and WSTG alignment (p=0.368). For Structural completeness, the Friedman test could not be calculated due to the lack of variability in the observations.

Only the Manual adjustments variable presented a statistically significant difference among the scenarios (Statistic=6.690, df=2, p=0.035). Kendall's coefficient of concordance (W=0.418) indicates a moderate effect size, suggesting that the amount of manual adjustments varied among the experimental scenarios.

4.4 Pairwise Comparisons

Table 6 and Table 7 present the results of the paired comparisons performed among scenarios S1, S2, and S3. For the ordinal and count variables, the p-values, the values adjusted by Holm's method, the difference between the medians, and the effect size r are reported, when applicable. For WSTG alignment, the results of the paired McNemar's test are presented, including the p-values, the values adjusted by Holm's method, and the statistics associated with the test.

Table 6: Pairwise comparisons for ordinal and count variables.

Variable	Comparison	p	p_Holm	Median_dif	r
Technical correctness	S1 vs S2	1,000	1,000	0,00	0,000
Technical correctness	S1 vs S3	0,346	1,000	0,00	0,667
Technical correctness	S2 vs S3	1,000	1,000	0,00	0,000
Practical reproducibility	S1 vs S2	–	–	0,00	–
Practical reproducibility	S1 vs S3	1,000	1,000	0,00	0,000
Practical reproducibility	S2 vs S3	1,000	1,000	0,00	0,000
Structural completeness	S1 vs S2	–	–	0,00	–
Structural completeness	S1 vs S3	–	–	0,00	–
Structural completeness	S2 vs S3	–	–	0,00	–
Procedural clarity	S1 vs S2	1,000	1,000	0,00	0,000
Procedural clarity	S1 vs S3	1,000	1,000	0,00	0,000
Procedural clarity	S2 vs S3	–	–	0,00	–
Manual adjustments	S1 vs S2	0,075	0,151	-2,50	0,672
Manual adjustments	S1 vs S3	0,354	0,354	1,00	0,327
Manual adjustments	S2 vs S3	0,036	0,107	2,00	0,858

Table 7: Pairwise comparisons for WSTG alignment.

Variable	Comparison	Test	p	p_Holm	b	c
Alignment WSTG	S1 vs S2	Paired McNemar	1,000	1,000	0	1
Alignment WSTG	S1 vs S3	Paired McNemar	1,000	1,000	0	1
Alignment WSTG	S2 vs S3	Paired McNemar	–	–	0	0

After applying Holm's correction, none of the paired comparisons remained statistically significant. The comparisons involving Manual adjustments presented values of p = 0.075 (S1 and S2) and p = 0.036 (S2 and S3), but they ceased to be significant after the p_Holm adjustment. The remaining variables did not present relevant differences among the scenarios.

Taken together, the results do not provide statistically consistent evidence that contextual enrichment with CWE/WSTG or with a local code snippet improved the technical correctness, practical reproducibility, structural completeness, procedural clarity, or alignment with the WSTG of the generated procedures. The main quantitative finding is that, under the evaluated scenarios, the increase in informational context was not sufficient to produce fully executable verification procedures.

5 Discussion

This section discusses the main findings of the experiment regarding the research question and the formulated hypotheses. The objective of the study was to analyze whether contextual enrichment with CWE descriptions, OWASP WSTG procedures, and local code snippets could improve the capability of an LLM to transform SAST alerts into reproducible verification procedures. Overall, the results do not support a statistically consistent improvement among the

evaluated scenarios. This finding suggests that the problem of operationalizing SAST alerts is not resolved solely by expanding the informational context provided to the model.

Answering the Research Question

RQ: To what extent do CWE/WSTG knowledge and local code influence LLM-generated security verification procedures from SAST alerts?
The results indicate that contextual enrichment alone was not enough to improve the generated procedures. CWE and WSTG documents retrieved via RAG did not significantly improve technical correctness, procedural clarity, or WSTG alignment. Thus, H_1 ("*the use of CWE and WSTG improves technical correctness, procedural clarity, and alignment with the corresponding WSTG procedure when compared to the isolated use of the SAST alert*") was not supported.
Local code did not significantly improve practical reproducibility. The generated procedures remained difficult to execute across all scenarios, with low reproducibility scores. Thus, H_2 ("*the inclusion of the local code snippet reduces the number of manual adjustments and improves the practical reproducibility of the generated procedures when compared to the other conditions*") was not supported for reproducibility.
A limited positive signal was observed for manual adjustments: the scenario with CWE/WSTG and local code required the lowest median number of adjustments. However, this effect did not remain significant after Holm correction, so it should be treated as exploratory.
These findings show that more context does not automatically produce reproducible verification procedures. LLMs may support initial drafting, but reliable SAST alert operationalization still requires human review and execution-based validation.

5.1 Contextual Enrichment Did Not Guarantee Operationalization

This study aims to analyze to what extent contextual enrichment with CWE descriptions, OWASP WSTG procedures, and local code snippets could improve the capability of an LLM to transform SAST alerts into reproducible verification procedures. To this end, three scenarios were compared, ranging from the isolated use of the alert to the combination of the alert, structured knowledge, and the code snippet associated with the issue reported by SonarQube.

The obtained results do not support that this enrichment produced a statistically significant improvement in the quality of the generated procedures. The global analyses did not identify significant differences among S1, S2, and S3 for technical correctness (p=0.223), practical reproducibility (p=0.368), procedural clarity (p=0.368), and alignment with the WSTG (0.368). Furthermore, practical reproducibility presented a median equal to 0 across the three scenarios, indicating that, in most cases, the produced procedures could not be successfully executed based on the artifacts generated by the LLM.

Consequently, this result suggests that transforming a SAST alert into an executable procedure requires more than expanding the informational context of the prompt. Although knowledge retrieval may support the identification of the vulnerability and its classification, the operationalization of the alert depends on a more precise procedural translation, capable of converting technical descriptions into actionable, ordered, and verifiable steps in the execution environment.

5.2 CWE/WSTG Can Increase Context, but Also Aperational Complexity

The findings reinforce the distinction between conceptual security knowledge and the practical operationalization of the test. Knowing the vulnerability's category, formal description, and WSTG reference procedure does not, in itself, imply the capability to produce an executable sequence of actions within the context of the target application.

In this sense, CWE and WSTG contribute to naming, framing, and contextualizing the vulnerability. This helps explain why the enriched scenarios did not present a statistically consistent improvement in variables such as technical correctness, procedural clarity, and alignment with the WSTG. The enrichment may have aided the description of the flaw at a conceptual level; however, this gain did not translate into an improvement in the capability for practical execution.

The variable that best highlights this limitation is practical reproducibility. Furthermore, it presents a median equal to 0 across all scenarios and has the lowest inter-rater agreement (AC2=0.242; observed agreement=54.2%), which indicates greater difficulty in applying the judgment and higher sensitivity to concrete operationalization failures. This result suggests that the main difficulty of the generated procedures was guiding the reproduction precisely enough to dispense with external complementation.

5.3 Local Code Appears To Help, but Does Not Resolve the Issue

Among all evaluated variables, the "Manual adjustments" variable was the only one that presented a significant global difference among the experimental scenarios in the Friedman test. Consequently, S3 presented the lowest median of manual adjustments (6.0), while S1 presented a median of 7.0, and S2 presented the highest median (8.5). The comparison between S2 and S3 presented a p-value of 0.036 (p=0.036) before the correction, but it ceased to be significant after Holm's adjustment.

Despite the absence of statistical confirmation after correction, this pattern can be interpreted as a preliminary sign that the local code contributes to reducing the adaptation effort required during execution. The lower median observed in S3 suggests that the code snippet associated with the alert may provide additional clues in order to make the procedure less generic and more tailored to the target. This effect points to a partial reduction in the interpretation and complementation work required during execution.

However, the code alone does not provide sufficient dynamic context to resolve the operationalization problem. Although it may expose the function or the logical block related to the alert, the execution of a verification test also depends on information that

frequently goes beyond the local snippet, such as application state, flow between components, authentication requirements, and other execution-context details. Thus, the local code appears useful as a complementary signal, but insufficient as an exclusive basis for the autonomous generation of executable procedures.

5.4 Low Reproducibility Is the Most Important Practical Finding

Despite the low practical reproducibility observed in the three scenarios, the high results for technical correctness, structural completeness, and procedural clarity indicate that LLMs were capable of understanding and structuring relevant information about the vulnerabilities. This behavior is aligned with the findings of Mao et al. [10], who demonstrated the efficacy of LLMs in generating vulnerability explanations, including their cause, location, and repair suggestions. Thus, the results of this study do not invalidate the use of LLMs in the operationalization of SAST alerts, but they indicate that their capability to produce fully reproducible procedures remains limited.

In light of this, a more suitable role for LLMs in this context is to support the initial elaboration of procedures, acting as drafting tools for security analysts. This behavior aligns with the literature that positions LLMs as support tools for code analysis, review, and debugging, as well as assistants integrated into development and security workflows [17, 19].

The results also highlight the need for additional mechanisms to guarantee the reliability and executability of the generated procedures. The low practical reproducibility observed indicates that the isolated use of LLMs is not sufficient to ensure operational consistency. The literature has pointed out that integration with instrumented execution, automated testing, and static analysis contributes to improving the validation of the generated outputs [2]. Even so, despite the support of external tools, such approaches do not guarantee the complete autonomy of the model, acting as verification mechanisms rather than substitutes for the analysis process [5]. These results reinforce the need for hybrid pipelines with explicit validation for security applications.

The illustrative artifact presented in Table 2 helps contextualize this difficulty. The S3 procedure for `typescript:S1523` was more specific than S1 and S2 because it referenced concrete code elements such as `body.orderLinesData`, `safeEval`, and `vm.runInContext`. Still, its execution depended on runtime information not fully encoded in the artifact, including route reachability, challenge activation, request structure, and observable error behavior. This helps explain why practical reproducibility was harder to judge than criteria based mainly on textual structure or vulnerability classification.

6 Threats to Validity

This section discusses the main threats to the validity of the study using the Wohlin et al. [18]'s categorization. It also describes the measures adopted to mitigate them throughout the experimental design and data analysis.

Construct validity. The quality of the generated procedures was operationalized by multiple variables: technical correctness, practical reproducibility, structural completeness, procedural clarity, alignment with WSTG, and manual adjustments. Due to the fact that part of these variables depends on human evaluation, there is a risk of subjectivity and partial overlap among constructs. To mitigate these issues, prior synchronization of the judges, blind evaluation, structured evaluation documents (golden set), and re-evaluation in case of divergences among the judges were adopted.

The fact that the golden set was visible to the judges during the assessment also poses a threat to construct validity. Although this material is useful for reducing ambiguity in the interpretation of expected results, it may have influenced the assessment of practical reproducibility. In particular, access to this reference might have mitigated difficulties encountered during execution based on the model-generated procedure, thereby reducing the independence of the judgment regarding the artifact's ability to guide the reproduction of the vulnerability without assistance. Consequently, the results should be interpreted as stemming from a reference-aided assessment rather than as a pure measure of an evaluator's ability to carry out the verification procedure without external support.

Internal validity. The repeated measures design using the same eight issues across the three experimental scenarios could introduce learning and familiarity effects. To mitigate such risks, the artifacts were anonymized and randomized, and each execution performed by the judges was conducted in an isolated environment. Nevertheless, the limitation arising from the model's variation remains, even with fixed parameters and multiple executions per scenario.

Furthermore, the limitation arising from the stochastic variability generated by LLMs remains. In the study, a single final response was evaluated for each issue-scenario combination, keeping the model, the generation parameters, and the prompt structuring constant. Although the use of the same model with fixed parameters contributes to increasing experimental control, the possibility remains that part of the variation in the quality of the generated procedures stems from the model's probabilistic behavior, and not exclusively from the differences among the scenarios.

External validity. External validity is limited by the use of a single target application, the OWASP Juice Shop, a single SAST tool, SonarQube, and a single LLM model. Thus, the results must be interpreted as evidence in a controlled environment, requiring replications with other environments, tools, models, and evaluator profiles to broaden their generalization.

Conclusion validity. A potential threat is the sample size used in the statistical analysis across the experimental scenarios. Although the sample was relatively small, this characteristic guided our choice of appropriate statistical tests. We carefully verified that the assumptions of the selected tests were satisfied to support the validity of the analysis.

7 Conclusion

This study evaluates to what extent different context sources influence the generation of security verification procedures by LLMs from SAST alerts. Three scenarios were compared: isolated SAST alert, alert enriched with CWE/WSTG documents retrieved via RAG, and alert enriched with CWE/WSTG augmented by a local code snippet. The procedures were evaluated by judges based on criteria of technical correctness, practical reproducibility, structural completeness, procedural clarity, alignment with WSTG, and manual adjustments.

The results do not provide consistent statistical evidence that contextual enrichment improved the quality of the generated procedures. Practical reproducibility remained low across all three scenarios, indicating that the artifacts produced by the LLM were rarely sufficient to support the reproduction of the vulnerability in the execution environment. This finding shows that the transition from a SAST alert to a verifiable procedure requires more than the inclusion of technical documentation in the prompt.

The main implication of the study is that LLMs should not be positioned as autonomous generators of reproducible security verification procedures. Their most defensible use lies in the initial elaboration of drafts, alert organization, and retrieval of related knowledge. The confirmation of the vulnerability still requires human review, controlled execution, and explicit validation mechanisms.

The study also suggests that the local code snippet may reduce part of the adaptation effort required from the evaluators, as scenario S3 presented the lowest median of manual adjustments. This result, however, must be treated as exploratory, since the paired comparisons did not remain significant after correction for multiple comparisons. Future studies should investigate whether richer contexts, including execution flow, application state, routes, credentials, input data, and test oracles, increase the reproducibility of the generated procedures.

As next steps, we recommend to expand the experimental corpus, include other SAST tools, evaluate different LLM models, and compare generation strategies based on prompting, RAG, and instrumented execution. It is also necessary to separate independent evaluations from reference-assisted evaluations, allowing a more precise measurement of how much of the generated procedure is actually executable without external support. These advances can support the construction of hybrid pipelines in which LLMs generate verification hypotheses, but operational validity is confirmed by execution, evidence, and observable criteria.

Ethical Considerations on the Use of AI

We used Large Language Models for language refinement, but all analysis and interpretation remain our sole responsibility.

Artifact Availability

We provide the following resources: a survey, anonymous data and environment at the following link: https://github.com/ASTRID-RESEARCH/gold-set-sast-for-llm/

Acknowledgments

This work was developed with support from the National Council for Scientific and Technological Development - CNPq (Process 312173/2025-3), Ceará State Foundation for the Support of Scientific and Technological Development (FUNCAP) (Process No. BMD-0008-03099.01.06/26) and This study was financed in part by the Brazilian Federal Agency for Support and Evaluation of Graduate Education – CAPES – Finance Code 001

References

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

[2] Greta Dolcetti, Vincenzo Arceri, Eleonora Iotti, Sergio Maffeis, Agostino Cortesi, and Enea Zaffanella. 2026. Ajudando LLMs a melhorar a geração de código usando feedback de testes e análise estática. *Discov. Artif. Intell.* 6, 1 (March 2026).

[3] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2026. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. *ACM Transactions on Software Engineering and Methodology* (2 2026). doi:10.1145/3797277

[4] Rasmus Ingemann Tuffveson Jensen, Vali Tawosi, and Salwa Alamir. 2024. Software Vulnerability and Functionality Assessment using Large Language Models. In *Proceedings of the Third ACM/IEEE International Workshop on NL-based Software Engineering* (New York, NY, USA). ACM, 25–28. doi:10.1145/3643787.3648036

[5] Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When Can LLMs Actually Correct Their Own Mistakes? A Critical Survey of Self-Correction of LLMs. *Transactions of the Association for Computational Linguistics* 12 (11 2024), 1417–1440. arXiv:https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl$_{a0}$0713/2478635/$tacl_{a0}$0713.pdf doi:10.1162/tacl_a_00713

[6] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[7] 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* (New York, NY, USA). ACM, 921–933. doi:10.1145/3611643.3616262

[8] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *International Conference on Learning Representations*, Vol. 2025. 35735–35758.

[9] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA). ACM, 544–555. doi:10.1145/3533767.3534380

[10] Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2025. Towards Explainable Vulnerability Detection With Large Language Models. *IEEE Transactions on Software Engineering* 51, 10 (2025), 2957–2971. doi:10.1109/TSE.2025.3605442

[11] OWASP Foundation 2021. *OWASP Top 10:2021*. OWASP Foundation. https://owasp.org/Top10/

[12] Rafael Ramires, Sarmad Bashir, Abbas Khan, Mehrdad Saadatmand, and Ibéria Medeiros. [n. d.]. *Friends or Foes? Combining Static Analysis Tools and LLMs for Vulnerability Detection*. Technical Report.

[13] Cathrin Schachner and Jasmin Wachter. 2026. Can AI Lower the Barrier to Cybersecurity? A Human-Centered Mixed-Methods Study of Novice CTF Learning. arXiv:2602.18172 [cs.CR] https://arxiv.org/abs/2602.18172

[14] Jiahao Shi and Tianyi Zhang. 2026. RESCUE: Retrieval Augmented Secure Code Generation. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=gbxhesw4UH

[15] Norbert Tihanyi, Yiannis Charalambous, Ridhi Jain, Mohamed Amine Ferrag, and Lucas C. Cordeiro. 2025. A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. 136–147. doi:10.1109/AST66626.2025.00020

[16] 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 Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[17] Jiayimei Wang, Tao Ni, Wei-Bin Lee, and Qingchuan Zhao. 2025. A Contemporary Survey of Large Language Model Assisted Program Analysis. arXiv:2502.18474 [cs.SE] https://arxiv.org/abs/2502.18474

[18] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2024. *Experimentation in Software Engineering*. Springer Berlin Heidelberg. doi:10.1007/978-3-662-69306-3

[19] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. 2025. Large Language Models for Cyber Security: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2025). doi:10.1145/3769676 Just Accepted.

[20] Mengyao Zhao, Kaixuan Li, Lyuye Zhang, Wenjing Dang, Chenggong Ding, Sen Chen, and Zheli Liu. 2025. A Systematic Study on Generating Web Vulnerability Proof-of-Concepts Using Large Language Models. arXiv:2510.10148 [cs.SE] https://arxiv.org/abs/2510.10148

[21] Orçun Çetin and Nazlı Bıyıklı. 2026. *Evaluating the Efficacy of Large Language Models for Automated Vulnerable Code Generation*. Springer, 197–212. doi:10.1007/978-3-032-17443-7_12