

Assessing Static Analysis Tools for Security Vulnerability Detection: An Empirical Study

Artur S. Farias
0009-0005-5576-124X

Computing Institute Federal University of Alagoas
in Maceio, Brazil
Email: asf2@ic.ufal.br

Rodrigo Rocha
0000-0003-1993-5044

Computing Institute Federal University of Alagoas
in Maceio, Brazil
Email: rodrigo.rocha@ufape.edu.br

Jamilson Dantas
0000-0001-9009-7659
Federal University of Pernambuco
in Recife, Brazil
Email: jrd@cin.ufpe.br

Abstract—Static analysis tools are widely adopted to support security vulnerability detection in modern software development, particularly when integrated into continuous integration (CI) pipelines. However, the increasing number of available tools and the operational constraints imposed by CI environments complicate tool selection, while empirical evidence supporting informed adoption decisions in realistic CI settings remains limited. This study addresses this gap by empirically assessing static analysis tools for security vulnerability detection in Java projects executed within GitHub Actions pipelines. The evaluation considers a selected set of representative open-source tools and analyzes their behavior across repositories containing documented, real-world vulnerabilities. The assessment focuses on detection effectiveness, coverage of vulnerability categories, consistency and complementarity across tools, and operational behavior in CI, including execution overhead and stability. The results reveal substantial variability among the evaluated tools, demonstrating that no single solution simultaneously optimizes detection capability, execution efficiency, and CI robustness. These findings expose explicit trade-offs that directly affect the practical adoption of static analysis in CI-driven development workflows. By providing CI-grounded empirical evidence, this study supports more informed tool selection and configuration decisions, benefiting researchers, practitioners, and organizations seeking to strengthen security-oriented practices in continuous integration environments.

Index Terms—Static Analysis, Security Vulnerability Detection, Empirical Study, Continuous Integration, Software Security

I. INTRODUCTION

SOFTWARE systems have become critical infrastructure across multiple domains, increasing the demand for reliability, quality, and security throughout the software development lifecycle. As these systems evolve in scale and complexity, security vulnerabilities remain a persistent concern, motivating organizations to strengthen assurance activities while maintaining rapid delivery. Industry reports consistently indicate that testing and quality assurance remain strategic priorities, with growing emphasis on automation and tool support in modern development pipelines [1], [2].

From a security perspective, the impact of vulnerabilities extends beyond technical risk and is reflected in economic indicators. Estimates on the cost of data breaches highlight the financial relevance of security failures and reinforce the need for preventive practices and automated controls throughout the software lifecycle [3]. Information security is commonly defined as the preservation of confidentiality, integrity, and availability of information assets [4], [5]. Within this context, software testing and analysis activities play a central role in supporting secure software development.

Software testing is defined as a systematic activity aimed at evaluating a system or component to identify defects and assess whether it satisfies specified requirements [6]. Among complementary assurance techniques, static analysis has gained prominence due to its ability to detect security-relevant issues without requiring program execution, enabling earlier feedback and supporting continuous development workflows [7]. In practice, static analysis tools are increasingly integrated into continuous integration (CI) pipelines, where they must operate under operational constraints such as build dependencies, limited execution time, and pipeline reliability requirements.

Despite their widespread adoption, selecting appropriate static analysis tools remains challenging. The growing number of available solutions, combined with differences in detection strategies, rule sets, and integration requirements, increases uncertainty for practitioners and organizations. Qualitative evidence from industry indicates that traditional evaluation practices frequently overlook false negatives and operational constraints that are critical for practical adoption [8]. Practitioner-oriented syntheses further suggest that tool selection is often influenced by popularity and perceived maturity, while systematic and transparent selection criteria are inconsistently applied [9]. Empirical studies also report that static application security testing (SAST) tools may produce heterogeneous warning profiles and substantial false-positive rates, negatively affecting trust and adoption [10], [11].

Prior research has investigated static analysis and security testing tools, including comparative evaluations of SAST tools for Java [12]. However, existing evidence provides limited insight into how these tools behave when operationalized in real-world CI environments, where integration robustness, execution overhead, and pipeline reliability are often decisive for adoption. The widespread adoption of CI and DevOps practices intensifies the need for actionable, context-aware evidence, as teams must balance security assurance with development velocity. Empirical observations of CI usage in open-source repositories further indicate that CI automation platforms such as GitHub Actions have become a common infrastructure for operationalizing quality and security checks [13].

This paper addresses this gap by presenting an empirical assessment of static analysis tools for security vulnerability detection in Java projects executed within continuous integration environments. The study evaluates a selected set of representative open-source tools integrated into GitHub Actions pipelines and applied to repositories containing real-world vulnerabilities. The assessment investigates four complementary dimensions: (i) detection effectiveness, (ii) consistency and complementarity of reported vulnerabilities across tools, (iii) execution overhead imposed on CI pipelines, and (iv) execution stability in real-world CI environments.

The results reveal substantial variability among tools across these dimensions, exposing trade-offs between detection capability, execution overhead, and pipeline robustness. These findings provide CI-grounded empirical evidence to support informed decision-making for tool selection and adoption in CI-driven development workflows.

The remainder of this paper is organized as follows. Section 2 presents related work and background concepts. Section 3 describes the empirical study design, including tool selection, evaluation metrics, and execution environment. Section 4 reports the results with respect to the research questions. Section 5 discusses implications for research and practice, along with threats to validity. Finally, Section 6 concludes the paper and outlines future work.

II. RELATED WORK

Software testing has long been recognized as a fundamental activity in software engineering, playing a central role in assessing software quality and identifying defects that may compromise system reliability and security [6]. As software

systems have evolved in size, complexity, and criticality, testing practices have continuously adapted to address new challenges, including shorter release cycles, distributed development, and increasing security requirements [18].

In parallel with the evolution of testing practices, security has emerged as a key quality attribute in modern software systems. Information security is commonly defined in terms of confidentiality, integrity, and availability of information assets [4]. Security vulnerabilities constitute a particular class of defects that may lead to severe technical and economic consequences if exploited [19]. As a result, security-oriented testing and analysis techniques have gained increasing attention in both academia and industry.

Static analysis techniques have been widely studied as an effective means to detect defects and vulnerabilities without requiring program execution [7]. Static Application Security Testing (SAST) tools specialize in identifying potential security vulnerabilities at the source code level and are commonly used to support secure software development [11], [10]. Empirical studies indicate that such tools differ substantially in terms of detection capability, precision, and types of reported vulnerabilities, which may affect their practical usefulness and adoption [12].

The adoption of continuous integration (CI) practices has further influenced how testing and analysis techniques are applied in practice. CI environments promote frequent integration and automated quality checks, enabling early feedback and faster development cycles [20]. Static analysis tools are increasingly integrated into CI pipelines as part of automated quality assurance processes [13]. However, CI environments impose operational constraints related to execution time, resource consumption, and pipeline stability, which may impact the effectiveness and feasibility of integrating security analysis tools into development workflows [21], [22].

Empirical software engineering research has emphasized the importance of systematic and context-aware evaluation of tools and techniques to support evidence-based decision-making [23]. Recent studies increasingly advocate the use of real-world projects and realistic execution environments to complement benchmark-based evaluations, providing insights into both technical effectiveness and practical adoption factors [24]. Building on this body of work, the following section reviews and compares existing empirical studies related to static analysis tools for security vulnerability detection.

The study entitled Comparison and Evaluation on Static

TABLE I
CONCISE COMPARISON OF RELATED WORK ON SAST EVALUATION.

Study	Evaluation context	Language	Primary focus	CI execution
Kaixuan et al[12]	Offline, benchmark-driven	Java	Vulnerability detection effectiveness	No
Wachiraphan et al[14]	Change-based secure code review	C/C++	Alert coverage on vulnerability-inducing changes	No
Esposito et al[15]	Controlled experimental benchmarks	Java	Precision-recall trade-offs in SAST	No
Dalaq et al[16]	Secondary synthesis (SLR)	Multiple	Evaluation practices and research gaps	Not evaluated
Ami et al[17]	Industry qualitative study	N/A	Practitioner concerns and adoption factors	Not evaluated
This study	CI pipelines (GitHub Actions)	Java	Detection signals and operational behavior	Yes

Application Security Testing (SAST) Tools for Java [12] aimed to analyze the effectiveness, detection capability, consistency, and performance of SAST tools by using well-defined criteria and real-world benchmarks. The authors selected seven free tools from a total of 161 existing tools in order to conduct an evaluation comparing them from different perspectives. The results demonstrated a low detection rate of real vulnerabilities by the tools, with a high number of vulnerabilities not being properly detected even when combining the detection capabilities of all the studied tools. In addition, real-world benchmark data containing 165 Java programs were generated. This information was obtained from large-scale benchmark tests using a mixture of proprietary benchmark repositories and real-world repositories. A limitation of the study was the lack of information about the tool collection process prior to the selection phase used to obtain the seven studied tools based on the defined criteria, which makes it impossible to replicate the selection process.

The study *An Empirical Study of Static Analysis Tools for Secure Code Review* [14] was conducted with the objective of investigating the ability of SAST tools to support secure code reviews, addressing three aspects: effectiveness, potential benefits, and waiting time. The study evaluated five SAST tools for projects written in C/C++, using a dataset of real code changes that contributed to exploitable vulnerabilities. It also quantified the potential benefits of prioritizing modified functions through SAST alerts. This analysis was based on a dataset comprising 319 real vulnerabilities originating from 815 commits that contributed to vulnerabilities across 92 projects. The results indicate that a single SAST tool can generate alerts on vulnerable changes in more than half of the commits that contribute to a vulnerability. Furthermore, combining alerts from multiple SAST tools can achieve an even higher vulnerability detection rate. Unlike studies that focused on repositories containing vulnerabilities, this study used 1,064 vulnerable files and 1,060 vulnerable functions across 815 commits.

A systematic literature review conducted by Dalaq et al. [16] synthesized existing empirical evidence on Static Application Security Testing (SAST) tools, with particular emphasis on evaluation metrics, benchmarks, challenges, and research gaps. Following a rigorous protocol based on Kitchenham and Charters' guidelines, the review analyzed 26 primary studies selected from multiple digital libraries. The authors observed a strong reliance on synthetic benchmarks, such as OWASP Benchmark and Juliet Test Suite, as well as considerable heterogeneity in experimental designs and reported metrics across studies. The review highlights that commonly used metrics, including true positive rate and false positive rate, are often reported inconsistently, which limits comparability among studies. Moreover, the findings reinforce that no single SAST tool consistently outperforms others across different vulnerability types, programming languages, and datasets. The authors also note that several studies report satisfactory performance on synthetic datasets but substantially lower detection rates when real-world vulnerabilities are considered. A key

conclusion of the review is the lack of empirical studies evaluating SAST tools under realistic execution conditions, particularly within CI/CD pipelines. While tool integration into CI environments is frequently discussed conceptually, empirical evidence addressing operational constraints, such as execution time, pipeline stability, and scalability, remains limited, indicating an important gap in the literature.

An empirical study [15] evaluated the effectiveness of Static Application Security Testing (SAST) tools under controlled experimental conditions, focusing on benchmark-driven assessments. The study relied primarily on synthetic datasets to systematically measure classical performance metrics such as precision, recall, and false positive rate, enabling consistent comparisons across tools while minimizing environmental variability. The results reported by the authors reinforce well-known trade-offs in SAST evaluations, showing that tools achieving higher recall often incur substantially higher false positive rates [15]. These findings corroborate previous empirical evidence that highlights the difficulty of simultaneously optimizing sensitivity and precision in static security analysis tools. Although methodologically rigorous, the study explicitly limits its scope to controlled experimental settings and does not consider the execution of SAST tools within continuous integration environments [15]. Consequently, aspects related to automated execution frequency, pipeline stability, execution time constraints, and integration overhead remain unexplored, restricting the applicability of the results to CI-based software development workflows.

A high-impact study published at a leading security venue [17] investigated advanced vulnerability detection techniques with a strong emphasis on technical depth and methodological rigor. The work analyzed security analysis approaches using complex datasets and detailed analytical methods, providing insights into fundamental challenges faced by static and hybrid vulnerability detection techniques. The findings demonstrate that even state-of-the-art approaches struggle to consistently detect vulnerabilities across diverse and realistic scenarios, revealing inherent limitations in current security analysis technologies [17]. By focusing on in-depth technical evaluation, the study contributes to a deeper theoretical understanding of vulnerability detection effectiveness. However, the study does not aim to assess the operational feasibility of deploying such techniques in modern development environments [17]. In particular, aspects related to continuous integration, large-scale automation, and tool behavior under CI execution constraints are outside the scope of the analysis, limiting its direct applicability to CI-oriented software engineering practice.

Table I summarizes the evaluation contexts adopted in prior studies on SAST effectiveness. Existing empirical evidence is predominantly derived from offline benchmark-based experiments [12], [15] or from change- and review-oriented analyses that do not consider automated execution environments [14]. Complementary secondary and qualitative studies highlight methodological heterogeneity and emphasize that false negatives, contextual relevance, and operational behavior are critical for practical adoption [16], [17]. However, none

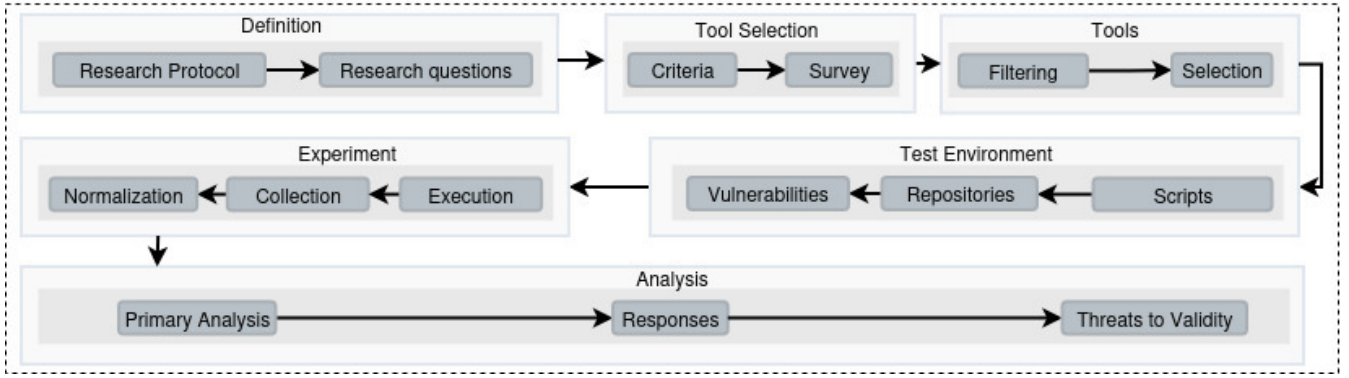


Fig. 1. Flow of the Empirical Study Stages.

of these studies empirically evaluates SAST tools under real continuous integration constraints. In contrast, the present study provides CI-grounded empirical evidence by executing SAST tools in GitHub Actions pipelines and analyzing both detection signals and operational properties relevant to modern CI-driven development workflows.

Taken together, existing empirical and secondary studies provide substantial evidence regarding the effectiveness and limitations of SAST tools in detecting software vulnerabilities. Prior work has explored benchmark-based evaluations [12], [15], secure code review support [14], systematic synthesis of evaluation practices [16], and technically rigorous security analyses [17]. Nevertheless, these studies predominantly assess SAST tools in isolated or controlled settings, without systematically addressing their execution under continuous integration constraints. In contrast, the present study complements existing work by empirically evaluating SAST tools within real CI pipelines based on GitHub Actions, focusing not only on detection effectiveness but also on operational aspects such as execution robustness, runtime behavior, and pipeline stability. By examining SAST tools under realistic automation conditions, this study addresses a gap in the literature and provides evidence that is directly relevant to practitioners adopting security analysis tools in modern CI-driven development workflows.

III. EMPIRICAL STUDY DESIGN

This section presents the design of the empirical study conducted to assess Java Static Application Security Testing (SAST) tools under Continuous Integration (CI) conditions. The study operationalizes each tool within GitHub Actions pipelines and evaluates their behavior when applied to repositories containing documented, real-world security vulnerabilities. The methodological design aims to support a fair, reproducible, and CI-oriented comparison by executing all tools under a common automation environment while systematically collecting analysis outputs and execution-related data. The phases can be seen in Figure 1.

The study is guided by the following overarching research question: **RQ0**: How effective and consistent are Java static

analysis tools in detecting security vulnerabilities when executed in GitHub Actions pipelines? To address RQ0, the empirical investigation is structured around four complementary sub-questions. **RQ1** examines the effectiveness of the tools in detecting known security vulnerabilities in the analyzed repositories. **RQ2** investigates the types of vulnerabilities reported, enabling a characterization of the security issues identified by each tool. **RQ3** analyzes the consistency of the results by assessing the overlap and divergence among tools in terms of detected vulnerabilities. Finally, **RQ4** focuses on the operational behavior of the tools in CI, considering aspects such as execution time, pipeline completion, and robustness when integrated into GitHub Actions workflows.

These research questions collectively frame the empirical protocol described in the remainder of this section, guiding the selection of tools, subject systems, execution procedures, and data analysis strategy.

A. Tool Selection Protocol

The tool selection process followed a multi-stage protocol designed to balance coverage, feasibility, and reproducibility. Initially, candidate tools were derived from the results of a prior systematic mapping study on software testing tools [25]. All 654 tools identified in the mapping were screened to determine whether they offered functionality related to security analysis, resulting in a filtered subset of 104 security-oriented tools.

Since the original mapping study focused on software testing in general and may not fully cover the landscape of Java security analysis tools, the filtered subset was complemented with prominent tools reported in a recent comparative study on Java SAST tools [12] totaling 110 tools. This step mitigates potential coverage bias and resulted in a consolidated candidate set of tools considered for empirical execution.

From this candidate set, tools were evaluated against six selection criteria: (i) native or fully functional support for Java; (ii) availability through free use or a free tier; (iii) command-line operability to enable automation; (iv) feasibility of execution within GitHub Actions pipelines, either via native actions, container-based execution, or direct CLI integration;

(v) explicit capability to detect security vulnerabilities using static analysis techniques; and (vi) evidence of recent maintenance activity (2024–2025), used as a preference criterion. Only tools satisfying the first five mandatory criteria were considered eligible for empirical execution, while the sixth criterion was used to prioritize among eligible candidates.

Applying these criteria resulted in a small set of feasible tools for CI-based assessment. The final selection comprises SonarQube, CodeQL, Semgrep, and Horusec. Although Spot-Bugs combined with FindSecurityBugs satisfied most technical criteria, it was excluded because its effective use requires project-specific configuration and tailoring. Such customization prevents a uniform pipeline design across repositories, undermining standardization, comparability, and reproducibility in a CI-based study. Horusec was selected instead due to its ability to support consistent automation across repositories using a standardized execution approach.

B. CI Pipeline Implementation

Each selected tool was integrated into a dedicated GitHub Actions workflow designed to minimize configuration complexity while ensuring correct execution. All pipelines were triggered on commits to the main branch and executed on GitHub-hosted runners. Each workflow included steps to retrieve the repository source code, install required dependencies and runtime environments, configure the tool, and execute the static analysis process.

To support uniform data collection, all workflows were configured to export analysis results in SARIF format whenever supported and to store these outputs as workflow artifacts. Although a common execution structure was adopted, tool-specific requirements prevented the creation of a single unified pipeline. CodeQL requires an additional compilation step for Java projects, Horusec relies on container-based execution using Docker, SonarQube requires Maven-based project builds, and Semgrep is installed and executed via `pip`. These differences reflect inherent tool design choices rather than methodological variation.

C. Subject Systems and Vulnerability Set

The subject systems consist of Java repositories containing documented security vulnerabilities. An initial search on GitHub using the query `"java vulnerability language:Java"` yielded 447 repositories. Repository documentation was manually inspected to identify projects that explicitly describe or enable the identification of existing security vulnerabilities, excluding repositories focused on tooling, tutorials, or theoretical examples.

This screening process resulted in an initial set of 13 repositories. For each repository, reported vulnerabilities were identified and cataloged, yielding a total of 27 distinct security vulnerabilities. While benchmark-oriented projects such as OWASP Benchmark and the Juliet Test Suite were initially considered, they were excluded from execution because they are not designed for direct CI execution and introduce build and packaging characteristics that hinder standardized pipeline

execution and result normalization. Consequently, the study focuses on repositories representing vulnerabilities occurring in realistic project structures.

Based on the number and diversity of vulnerabilities, 10 repositories were selected for empirical execution. This selection ensures sufficient vulnerability coverage while maintaining feasibility for detailed analysis under CI constraints.

D. Execution, Data Processing, and Analysis Strategy

Each selected tool was executed independently on each of the 10 subject repositories using its corresponding GitHub Actions workflow. Executions were triggered under identical CI conditions, ensuring that all tools ran on GitHub-hosted runners with comparable environment constraints. For each execution, the complete analysis output, execution time, and pipeline completion status were collected. Pipeline failures caused by tool errors, build issues, or unmet dependencies were explicitly recorded and considered as part of the operational assessment.

The analysis outputs produced by the evaluated tools exhibit substantial heterogeneity in terms of format, level of detail, and vulnerability taxonomy. To enable a meaningful comparison across tools, a systematic data extraction and normalization procedure was applied. Each reported finding was manually inspected and mapped to a vulnerability previously identified in the corresponding repository. To reduce subjectivity, the mapping and normalization process was independently reviewed and iteratively refined to ensure consistency across tools and repositories. When a reported issue could not be matched to an existing vulnerability category, a new category was created to represent that finding. This process resulted in the identification of three additional vulnerability categories not originally documented, yielding a total of 30 distinct vulnerability categories used as the analytical baseline.

This normalization step plays a central methodological role by harmonizing tool-specific outputs into a unified representation, enabling consistent analysis of detection coverage, overlap, and divergence. By abstracting away tool-specific nomenclature and reporting granularity, the study focuses on the presence or absence of vulnerability detection rather than on raw warning counts alone.

The analysis strategy is structured around the defined research questions. To address **RQ1**, detection effectiveness is assessed by examining the ability of each tool to identify vulnerabilities from the normalized set across repositories. **RQ2** is addressed by analyzing the distribution of detected vulnerabilities across categories, characterizing the types of security issues reported by each tool. To answer **RQ3**, consistency is examined by analyzing the overlap and divergence of detected vulnerabilities among tools for the same repositories. Finally, **RQ4** focuses on operational behavior in CI, considering execution time, pipeline completion status, and the volume of reported findings as indicators of practical feasibility in GitHub Actions environments.

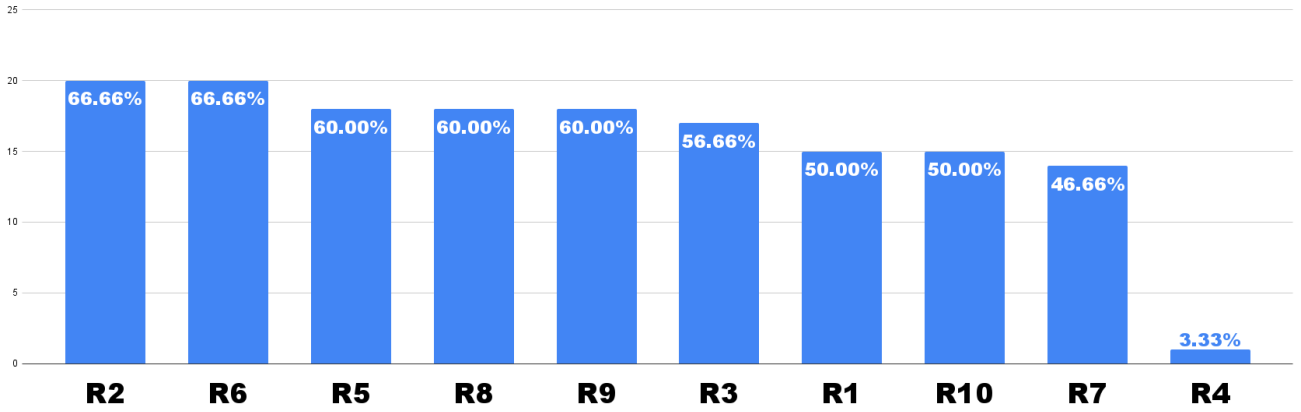


Fig. 2. Number of different types of flaws found per repository.

IV. EMPIRICAL RESULTS OF STATIC ANALYSIS TOOL EVALUATION

This section reports the empirical results of executing four static analysis tools (Horusec, Semgrep, CodeQL, and SonarQube) on ten Java repositories containing documented security vulnerabilities, operationalized through GitHub Actions. The results are organized to support evidence-based comparison in CI settings, covering: (i) the feasibility funnel for tool selection, (ii) the vulnerability landscape observed across subject systems, (iii) vulnerability detection outcomes by tool, (iv) cross-tool consistency, and (v) CI operational behavior. Overall, the results indicate that no single tool consistently dominates across detection effectiveness, operational overhead, and CI robustness, revealing explicit trade-offs that directly affect tool adoption in CI-driven environments.

A. Tool Selection Funnel and CI Feasibility

A screening process was applied to the tool set derived from the prior mapping, filtering candidates according to security relevance and practical constraints for CI execution. Figure 3 reports the number of tools meeting each selection criterion, illustrating the progressive reduction of candidates driven by Java support, command-line automation, and compatibility with GitHub Actions.

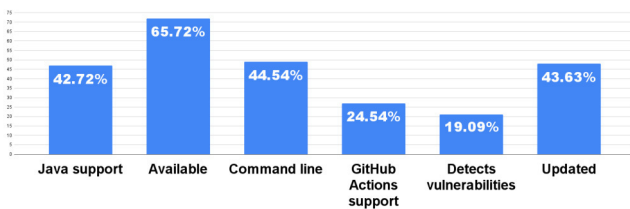


Fig. 3. Number of tools approved in each criterion.

The results indicates that, while a substantial portion of candidates provide Java support and free availability, less

than half are up to date, and only a minority offers CI-ready integration. This result operationalizes a key feasibility constraint: tool availability does not imply tool deployability in modern CI pipelines.

B. Vulnerability Landscape Across Repositories

The empirical execution considered a normalized set of 30 vulnerability categories observed across the selected repositories. Figure 2 reports the number of distinct vulnerability categories identified per repository, showing a concentration of repositories with similar vulnerability breadth and a small number of outliers.

Two repositories (WebGoat and Svja) exhibited the largest breadth of vulnerability categories, while Serverless-Goat-Java presented a markedly smaller footprint. This distribution suggests that repository characteristics influence both vulnerability footprint and analyzability under CI constraints.

C. Vulnerability Detection Outcomes and Tool Assertiveness (RQ1)

Detection outcomes were analyzed by relating repository-declared vulnerability presence to tool findings under four categories: (i) True Positive, (ii) False Negative, (iii) False Positive, and (iv) True Negative. Figure 4 summarizes these outcomes per tool.

Across all tools, False Negative concentrates the largest share of outcomes, indicating that many repository-vulnerability pairs are not instantiated in every repository context. For True Positive, CodeQL exhibits the highest number of correct detections, followed by Horusec and Semgrep, while SonarQube shows the lowest count. False Positive outcomes are highest for SonarQube and lowest for CodeQL, indicating different coverage and miss profiles across tools in this dataset.

To provide an aggregate Accuracy-oriented view, Figure 5 consolidates outcomes into correct detections, False Negatives, and False Positives. CodeQL presents the highest share of correct detections and the lowest False Negatives rate, while Semgrep presents the lowest False Positives rate. Horusec

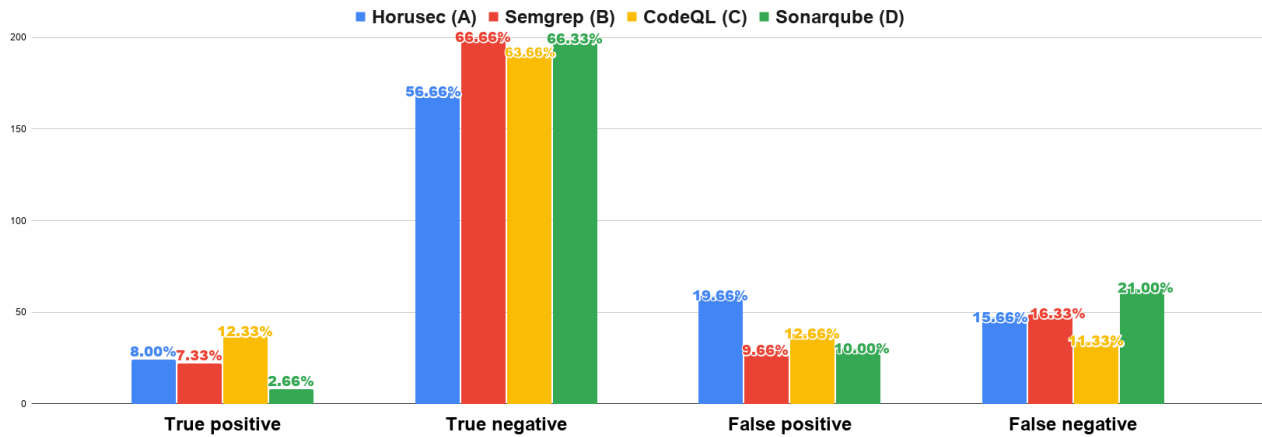


Fig. 4. Analysis of vulnerability detection by tool.

shows a higher False Positive incidence, and SonarQube shows the highest False Negative incidence among the evaluated tools.

D. Most Frequent Vulnerability Categories (RQ2)

Table II reports the frequency with which vulnerability categories were identified across the ten repositories. Classical vulnerability types such as SQL Injection, XSS, XXE, Path Traversal, and Sensitive Data Exposure appear across most repositories, indicating both prevalence in the selected subjects and maturity of rules for these categories.

The distribution also shows categories with moderate to low occurrence and categories that were not detected by any tool in the analyzed repositories, which may reflect either low prevalence in the subject systems or tool limitations for context-dependent categories.

E. Cross-Tool Consistency and Complementarity (RQ3)

Consistency was examined by analyzing how many tools co-detected the same vulnerability category in the same repository

context. Figure 7 summarizes the distribution by group size. A substantial fraction of cases are single-tool detections, and multi-tool consensus decreases sharply as group size increases, with full four-tool consensus being rare.

This pattern suggests complementarity rather than redundancy: multiple tools contribute unique detections, while only a small portion of detections are consistently replicated across tools.

F. Operational Behavior in GitHub Actions (RQ4)

Operational behavior was measured via GitHub Actions execution time and stability across repositories. Figure 6 reports execution time per repository and reveals systematic runtime differences among tools. Horusec and Semgrep exhibit consistently lower runtimes, whereas CodeQL remains the slowest tool across all repositories.

In addition to overhead, stability differences were observed. SonarQube did not complete analysis successfully for multiple repositories, yielding early completion without reported

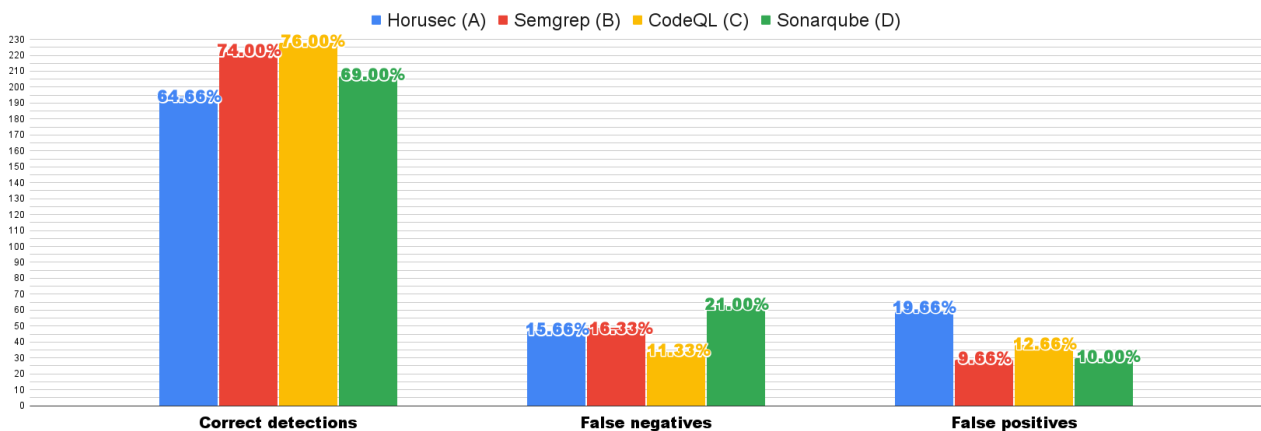


Fig. 5. Accuracy values across the tools.

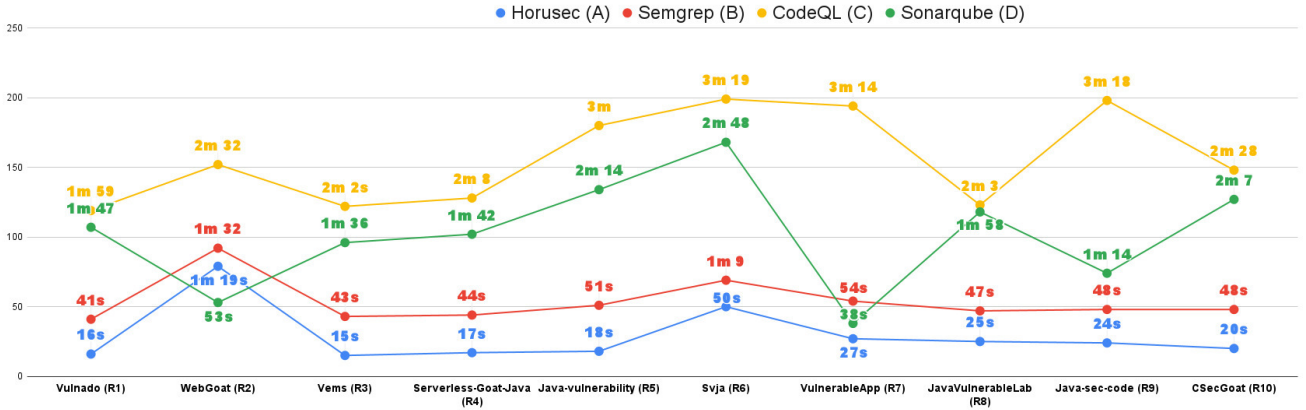


Fig. 6. Tool execution time per repository.

TABLE II
MOST FREQUENTLY FOUND VULNERABILITIES AMONG THE ANALYZED REPOSITORIES.

Vulnerability	Rep	Tools
Path Traversal	90%	5,26%
Sensitive Data Exposure	90%	6,07%
SQL Injection	90%	9,31%
XSS (Cross-Site Scripting)	90%	6,88%
CRLF Injection	80%	4,45%
Server-Side Request Forgery (SSRF)	80%	3,23%
XML External Entities (XXE)	80%	6,47%
Hardcoded Credentials	80%	3,64%
Remote Code Execution (RCE)	70%	4,85%
URL Redirect	70%	5,26%
Command Injection	60%	4,45%
Insecure Deserialization	60%	5,26%
Secure Cookie Flag	60%	4,45%
Security Misconfiguration	60%	2,83%
Weak Hashing	60%	4,04%
Cross-Site Request Forgery (CSRF)	50%	2,02%
Weak Cryptography	50%	3,23%
Weak Randomness	50%	2,02%
Denial of Service (DoS)	40%	1,61%
XPATh Injection	40%	2,42%
Broken Authentication	30%	1,21%
Vulnerable Components	20%	0,81%

findings, which impacts both result comparability and CI reliability.

To connect time overhead to output volume, Figure 8 reports findings-per-second as an efficiency indicator. Horusec presents the highest average findings-per-second, Semgrep presents moderate efficiency with stable behavior, SonarQube exhibits high variability influenced by incomplete runs, and

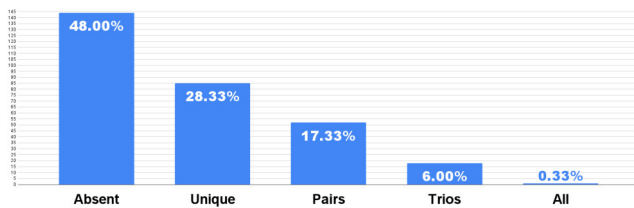


Fig. 7. Distribution of cases by tool group size.

CodeQL exhibits the lowest findings-per-second, indicating high time cost per reported finding in this CI setting.

V. ANALYSIS AND DISCUSSION

This section discusses the results reported in Section IV, emphasizing implications for the adoption of static analysis tools in CI-oriented security workflows. The discussion integrates evidence related to feasibility constraints for CI deployment, detection trade-offs across tools, complementarity and consistency of results, and the role of operational overhead and reliability as first-class adoption drivers.

The tool selection funnel (Figure 3) provides direct empirical evidence that the availability of security tools does not necessarily translate into deployability under CI constraints. Although a large number of tools offer Java support and free access, compatibility with GitHub Actions emerges as a decisive restriction, substantially narrowing the set of feasible candidates. This finding indicates that CI-oriented security programs must explicitly consider automation capability, integration maturity, and maintenance activity during tool selection. Otherwise, adoption is likely to incur high setup costs, brittle pipelines, or limited long-term sustainability. For teams operating under strict CI constraints, deployability and execution stability should therefore be treated as primary selection criteria rather than secondary considerations.

The detection outcome distributions (Figure 4) and the consolidated precision view (Figure 5) highlight clear trade-offs among the evaluated tools. CodeQL demonstrates the strongest balance for declared vulnerability detection, combining the highest proportion of correct detections with the lowest incidence of false negatives (Figure 5). This profile supports its adoption in scenarios where maximizing detection of known or documented vulnerabilities is critical, although at the cost of higher execution overhead in CI environments (Figure 6). If maximizing detection of known vulnerabilities is the primary objective, higher CI overhead may be acceptable, provided that pipeline time budgets are explicitly managed.

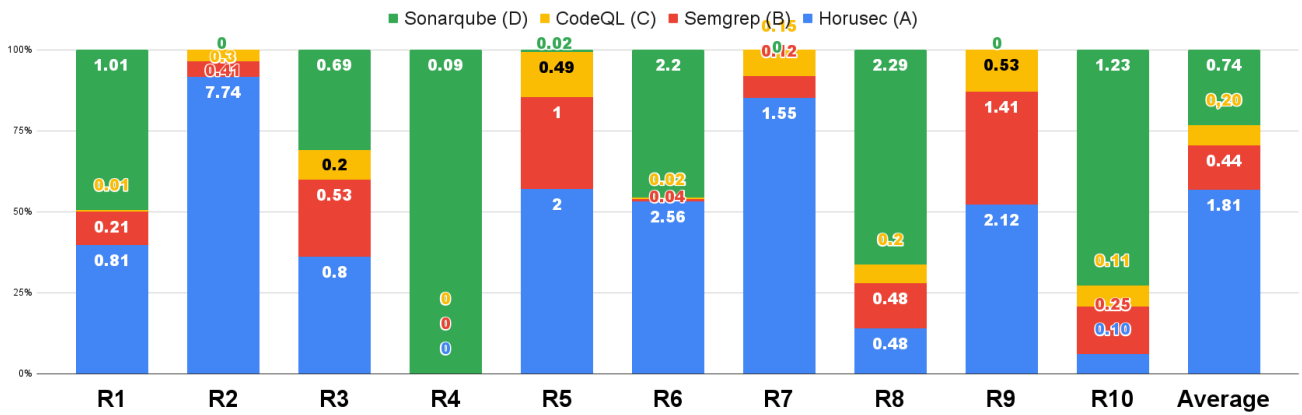


Fig. 8. Number of flaws detected per second during the analyses.

Semgrep presents a more conservative detection profile, achieving high correct-detection rates while maintaining the lowest false-positive incidence among the evaluated tools (Figure 5). This behavior can reduce triage effort and improve developer acceptance in CI pipelines. In contrast, Horusec exhibits a higher false-positive incidence and a stronger presence in non-declared detections (Figures 5 and 4), indicating greater sensitivity that may increase coverage but also raises the cost of result inspection. SonarQube shows the highest false-negative incidence and a lower number of declared-present detections, and its incomplete executions across multiple repositories further constrain interpretability and adoption (Figure 6). In CI contexts, this combination of missed detections and execution instability may reduce trust in the tool as a primary security gate. For CI pipelines used as quality or security gates, tools that exhibit execution instability may undermine confidence by producing incomplete or misleading assurance signals.

Analysis of vulnerability category coverage (Table II) shows that widely documented vulnerability families dominate the analyzed repositories. This observation is consistent with the maturity of static analysis rules for classical categories such as SQL Injection and Cross-Site Scripting, which remain both prevalent and detectable. At the same time, the absence of detections for certain categories suggests practical gaps, particularly for vulnerabilities that are context-dependent or semantically complex. These results reinforce that static analysis in CI should be treated as a critical security layer, but not as a complete substitute for complementary assurance techniques when broader coverage is required. This suggests that static analysis in CI should be complemented with additional assurance techniques when organizations require coverage of context-dependent or semantically complex vulnerabilities.

Consistency and complementarity analysis further indicates that single-tool detections account for a substantial portion of identified vulnerabilities, while full consensus across all tools is rare (Figure 7). This pattern provides empirical evidence of complementarity rather than redundancy: different tools

contribute unique detections. From a practical perspective, this suggests that multi-tool strategies can improve coverage, but must be balanced against additional operational costs. From a tooling strategy perspective, combining multiple tools can improve coverage, but only when the added operational cost is explicitly justified. From a research perspective, the results highlight the importance of reporting overlap structures in comparative evaluations, as overlap directly informs whether tools provide redundant or complementary value under CI constraints.

The strong overlap observed between CodeQL and Semgrep, as well as between CodeQL and Horusec (Figure 9), indicates a higher degree of convergence in vulnerability identification, reflecting either similar coverage profiles or complementary rule sets. In contrast, the limited presence of SonarQube in shared detection groups highlights its lower consistency and reduced contribution to common detections.

Operational overhead and reliability emerge as decisive adoption factors in CI settings. Runtime and stability evidence (Figure 6) shows systematic variation among tools: Horusec and Semgrep exhibit faster execution and more predictable pipeline impact, while CodeQL consistently imposes higher overhead. Efficiency results (Figure 8) further indicate that time investment does not translate uniformly into outputs, with Horusec converting runtime into reported findings more efficiently and CodeQL presenting the lowest findings-per-second profile. From an adoption standpoint, these results suggest that CI integration decisions should explicitly account for pipeline time budgets and failure modes. Tools that fail to complete analysis, as observed for SonarQube in multiple repositories, may produce misleading “clean” outcomes, undermining trust and potentially weakening security posture if interpreted as absence of vulnerabilities.

A. Threats to Validity

From an internal validity perspective, tool execution required tool-specific workflows and prerequisites, which may introduce execution asymmetries. In particular, SonarQube did

not complete analysis in multiple repositories (Figure 6); early completion without reported findings may bias comparisons by underreporting detections. Differences in build requirements, such as compilation prerequisites, may also affect both stability and detection opportunities.

Regarding construct validity, the evaluation relies on a normalized vulnerability set and on the alignment between repository documentation and the presence of vulnerabilities. Repository-declared vulnerabilities may be incomplete or inconsistently specified, affecting the interpretation of declared-present and declared-absent categories (Figure 4). In addition, tool outputs use heterogeneous taxonomies and reporting granularities, and the normalization process may introduce mapping subjectivity.

In terms of external validity, the subject systems consist of ten Java repositories and the execution environment is GitHub Actions. The results may not generalize to other programming languages, industrial codebases, or CI platforms. Variability in repository vulnerability footprints (Figure 2) suggests that tool behavior may differ under alternative project structures and dependency configurations. The study aims to provide CI-grounded empirical evidence and comparative insights rather than statistical generalization; therefore, observed trends should be interpreted as behavior within the analyzed CI setting and subject repositories.

VI. FINAL REMARKS

Static analysis tools have become an integral component of modern software development processes, particularly when embedded into continuous integration (CI) pipelines as part of security-oriented quality assurance practices. The widespread adoption of platforms such as GitHub Actions has intensified the need for automated security analysis solutions that operate reliably under CI constraints, providing timely feedback without compromising development velocity. In this context, Static Application Security Testing (SAST) tools are expected to combine detection capability with operational robustness and predictable execution behavior.

Despite their growing adoption, the selection and effective use of SAST tools in CI environments remain challenging. Tools differ substantially in detection strategies, vulnerability coverage, reporting profiles, and integration requirements, while CI pipelines impose constraints related to execution time, build dependencies, and stability. These factors force practitioners to balance security coverage against noise, performance overhead, and the risk of pipeline failures, often leading to adoption decisions based on popularity rather than CI-grounded empirical evidence. This concern aligns with recent qualitative industry findings highlighting false negatives and operational behavior as critical adoption factors [17].

This study addresses this gap by empirically assessing static analysis tools for security vulnerability detection in Java projects executed within GitHub Actions pipelines. An empirical study was conducted on ten repositories containing documented, real-world vulnerabilities. Tool candidates were filtered through a criteria-based selection process considering Java support, free availability, automation capability, CI compatibility, vulnerability detection focus, and maintenance activity. Four tools—Horusec, Semgrep, CodeQL, and SonarQube—were selected and evaluated using dedicated GitHub Actions workflows. Analysis outputs were normalized into a unified representation comprising 30 vulnerability categories, enabling consistent comparison of detection coverage, overlap, and divergence across tools.

The results reveal substantial variability among the evaluated tools, indicating that no single solution simultaneously optimizes detection effectiveness, operational efficiency, and CI robustness. CodeQL achieved the strongest detection performance at the cost of higher CI overhead, Semgrep maintained high correctness with low false-positive incidence and stable execution, and Horusec exhibited the most favorable operational behavior in terms of runtime efficiency. In contrast, SonarQube showed limitations in CI robustness, failing to complete analysis for a non-trivial portion of the repositories and potentially producing misleading outcomes. Consistency analysis further demonstrated limited multi-tool consensus,

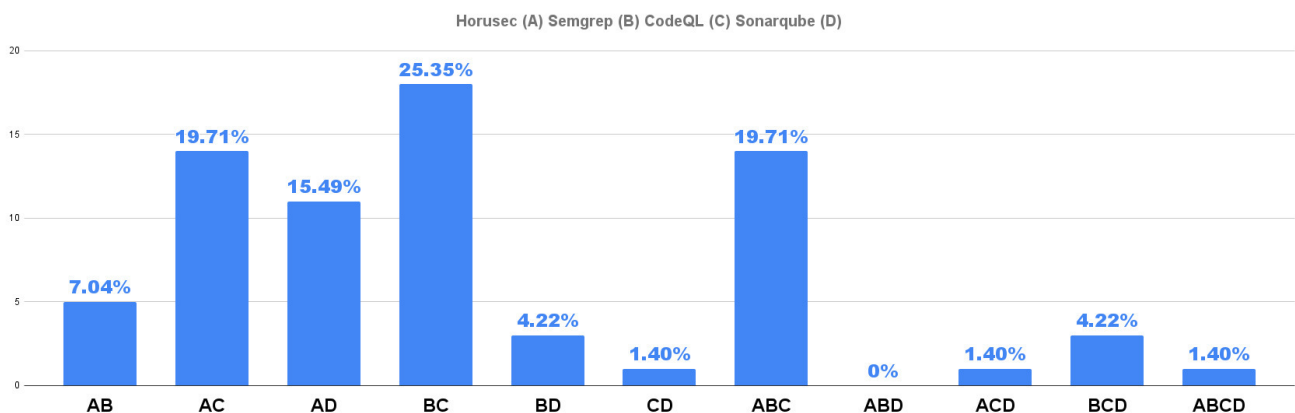


Fig. 9. Vulnerability Coverage Across Tools.

providing empirical evidence of complementary rather than redundant coverage.

From a decision-making perspective, these findings highlight that SAST adoption in CI-driven environments must be guided by explicit trade-offs rather than assumptions of universal effectiveness. Researchers benefit from CI-grounded empirical evidence on tool behavior and overlap structures, while practitioners and DevSecOps teams gain actionable insights to align tool selection with organizational priorities related to coverage, overhead, and reliability. Tool maintainers and platform integrators may leverage the observed limitations to improve automation support, execution stability, and reporting consistency.

Future work may broaden the set of subject repositories, evaluate additional tools and configuration strategies, and assess portability across alternative CI platforms. Incorporating additional operational metrics and execution strategies may further clarify how organizations can balance security effectiveness with CI efficiency in enterprise-oriented development workflows.

VII. ACKNOWLEDGMENT

Financial support for this research was provided by the CAPES Social Demand Program and the Federal University of Alagoas (UFAL). We also thank the UFAL Institute of Computing (IC) and the DKD Lab from the Federal University of the Agreste of Pernambuco (UFAPE).

Artificial intelligence was used to support the translation of this work into English.

REFERENCES

- [1] Capgemini. (2024) World quality report 16th edition | 2024-25. Acesso em: 31 jan. 2025. [Online]. Available: <https://ebook.capgemini.com/world-quality-report-2024-25/>
- [2] P. Wadhvani and A. Ambekar. (2025) Software testing market size. Acesso em: 31 jan. 2025. [Online]. Available: <https://www.gminsights.com/industry-analysis/software-testing-market>
- [3] Statista, “Cost of a data breach in the u.s. 2024,” Statista Research Department, 2024, accessed: 2025-03-05. [Online]. Available: <https://www.statista.com/statistics/273575/us-average-cost-incurred-by-a-data-breach/>
- [4] ISO/IEC27000, *Information technology — Security techniques — Information security management systems — Overview and vocabulary*, INTERNATIONAL STANDARD Std. ISO/IEC 27000:2018(E), 2018.
- [5] J. Holdsworth and M. Kosinski. (2024) What is information security? Acesso em: 5 mar. 2025. [Online]. Available: <https://www.ibm.com/think/topics/information-security>
- [6] ISO/IEC/IEEE., *International Standard—Software and Systems Engineering Software Testing—Part 1: General Concepts*, ISO/IEC/IEEE Std. 29119-1:2022(E), 2022.
- [7] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. USA: Cambridge University Press, 2016.
- [8] Y. Li, P. Yao, K. Yu, C. Wang, Y. Ye, S. Li, M. Luo, Y. Liu, and K. Ren, “Understanding industry perspectives of static application security testing (sast) evaluation,” *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. doi: 10.1145/3729404. [Online]. Available: <https://doi.org/10.1145/3729404>
- [9] P. Raulamo-Jurvanen, M. Mäntylä, and V. Garousi, “Choosing the right test automation tool: a grey literature review of practitioner sources,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 2017. doi: 10.1145/3084226.3084252 pp. 21–30.
- [10] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, “An empirical study of security warnings from static application security testing tools,” in *Journal of Systems and Software*, 2019. doi: 10.1016/j.jss.2019.110427 p. 110427.
- [11] J. Yang, L. Tan, J. Peyton, and K. A. Duer, “Towards better utilizing static application security testing,” in *41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019. doi: 10.1109/ICSE-SEIP.2019.00014 pp. 51–60.
- [12] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen, “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*, 2023. doi: 10.1145/3611643.361626 pp. 921–933.
- [13] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, “On the use of github actions in software development repositories,” in *2022 IEEE International Conference on Software Maintenance and Evolution*, 2022. doi: 10.1109/ICSME55016.2022.00029 pp. 235–245.
- [14] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, “An empirical study of static analysis tools for secure code review,” in *the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi: 10.1145/3650212.3680313 pp. 691 – 703.
- [15] M. Esposito, V. Falaschi, and D. Falessi, “An extensive comparison of static application security testing tools,” in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE ’24. New York, NY, USA: Association for Computing Machinery, 2024. doi: 10.1145/3661167.3661199. ISBN 9798400717017 p. 69–78. [Online]. Available: <https://doi.org/10.1145/3661167.3661199>
- [16] D. Dalaq, K. F. Daya, A. Dalaq, M. N. Arefin, and M. K. Niazi, “A systematic literature review on static application security testing (sast) tools: Evaluation, benchmarks, challenges, and future directions,” in *Proceedings of the 2025 29th International Conference on Evaluation and Assessment in Software Engineering Companion*, ser. EASE Companion ’25. New York, NY, USA: Association for Computing Machinery, 2025. doi: 10.1145/3727967.3756838. ISBN 9798400718328 p. 162–168. [Online]. Available: <https://doi.org/10.1145/3727967.3756838>
- [17] A. S. Ami, K. Moran, D. Poshvyanyk, and A. Nadkarni, ““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)*, 2024. doi: 10.1109/SP54263.2024.00019 pp. 3979–3997.
- [18] M. Kassab, J. F. DeFranco, and P. A. Laplante, “Software testing: State of the practice,” *IEEE Software*, vol. 34, pp. 46–52, 2017. doi: 10.1109/MS.2017.3571582
- [19] G. Weidman, *Penetration Testing: A Hands-On Introduction to Hacking*, 1st ed. No Starch Press, 2014.
- [20] I. Sommerville, *Software Engineering*, 10th ed. São Paulo: Pearson Universidades, 2019.
- [21] S. Romano, F. Zampetti, M. T. Baldassarre, M. D. Penta, and G. Scanniello, “Do static analysis tools affect software quality when using test-driven development?” in *ESEM ’22: Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022. doi: 10.1145/3544902.3546233 pp. 80–91.
- [22] J. Santos, D. A. da Costa, and U. Kulesza, “Investigating the impact of continuous integration practices on the productivity and quality of open-source projects,” in *ESEM ’22: Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2022. doi: 10.1145/3544902.3546244 pp. 137–147.
- [23] B. Kitchenham and S. M. Charters, “Guidelines for performing systematic literature reviews in software engineering,” Keele University and Durham University, Tech. Rep. EBSE-2007-001, January 2007. [Online]. Available: https://www.elsevier.com/_data/promis_misc/525444systematicreviewsguide.pdf
- [24] F. Pudlitz, F. Brokhausen, and A. Vogelsang, “What am i testing and where? comparing testing procedures based on lightweight requirements annotations,” *Empirical Software Engineering*, vol. 24, no. 4, pp. 2809–2843, 2020. doi: 10.1007/s10664-020-09815-w
- [25] A. Farias, R. Rocha, I. Vanderlei, J. Araújo, A. Araújo, and J. Dantas, “Software testing evidence: Results from a systematic mapping,” in *Proceedings of the 21st International Conference on Web Information Systems and Technologies - Volume 1: WEBIST, INSTICC*. SciTePress, 2025. doi: 10.5220/0013739100003985. ISBN 978-989-758-772-6 pp. 386–393.