Comparative Analysis of Static Application Security Testing Tools on Real-world Java Vulnerabilities

Wilmer Ansgariusson
wi8206an-s@student.lu.se

Jonathan Ståhl
jo1466st-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Christian Gehrmann

Examiner: Thomas Johansson

May 25, 2025

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

With the increasing complexity and scale of modern software systems, ensuring software security is more critical than ever. As projects grow, so does the likelihood of vulnerabilities being introduced. Static Application Security Testing (SAST) tools assist developers in identifying such vulnerabilities during development. In this study, five Java SAST tools (*Bearer*, *CodeQL*, *Horusec*, *Semgrep* and *SonarQube*) were evaluated based primarily on their vulnerability detection rate and execution time to determine their effectiveness.

The tools were tested using Java entries from the CVEfixes data set, which contains real-world code changes, including both pre- and post-fix versions of known vulnerabilities. Each tool analyzed code before and after vulnerability fixes across three scenarios, varying the context available (single files, modified files and full projects). True positives were defined as vulnerabilities detected before, but not after, a fix. This approach helps assess a tool's ability to correctly identify actual vulnerabilities. Among the tools, **CodeQL** and **Horusec** performed the best, with CodeQL showing stronger potential if allowed to be utilized fully. Bearer underperformed, while SonarQube and Semgrep may offer better results with more permissive configurations.

Popular Science Summary

Fast, Smart, or Just Loud? We Put Security Scanners to the Test

Application security is more critical than ever, as software becomes increasingly central to our everyday lives. Insecure applications can lead to data theft, financial loss, or system compromise. To address these risks, developers rely on various tools to catch bugs early, one common approach being Static Application Security Testing (SAST). We evaluated and compared several SAST tools to assess their effectiveness in identifying security issues during development.

A SAST tool is a type of computer program that can locate and identify security bugs in code. There are a lot of different SAST tools available on the market and it can be difficult to know which one to use. Interviews reveal that programmers choose such tools based on popularity rather than performance. Programmers also do not trust SAST tool benchmarks, stating that they contain bugs that would not occur in real-world software.

We want programmers to be able to trust benchmarks and therefore conducted our own tests on SAST tools. We scanned five popular SAST tools on just under 500 real security bugs from publicly available software projects.

Our results show that the best performing SAST tools were CodeQL and Horusec. We could also conclude that it is difficult to determine an actual winner since results can vary a lot based on how the tools are configured. The tools detected only a small portion of the actual bugs in the programs and often sounded the alarm incorrectly. This indicates that while SAST tools are helpful, solely relying on them would be a mistake. As AI continues to advance, it might one day offer a smarter alternative to today's SAST tools, but more research is needed.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

In today's software-driven economy, vulnerabilities are a persistent and costly threat. These vulnerabilities can be exploited to steal user data, disrupt services or compromise intellectual property, posing significant risks for financial and reputational consequences. With privacy regulations such as the General Data Protection Act (GDPR) and the Health Insurance Portability and Accountability Act (HIPAA), safeguarding user data is more important than ever in order to avoid legal repercussions or public backlash. Similarly, service disruptions can lead to lost revenue and distrust, while breaches involving sensitive data or trade secrets can have long-term strategic repercussions.

Catching and fixing security vulnerabilities early is essential and cost-effective. It has been estimated that fixing a vulnerability during production is roughly 30 times more expensive than during development [42]. One tool enabling such early detection of vulnerabilities is Static Application Security Testing (SAST). By analyzing source code or compiled binaries without executing them, SAST can identify security vulnerabilities at the very start of the software development life cycle. Integrated in the development environment or deployed as part of the CI/CD pipeline, vulnerabilities can be detected and fixed before ever reaching production.

As with any security tool, SAST should be used with caution; it is easy to blindly rely on such tools as a cheap label of security. In practice, no SAST tool is perfect and performance varies significantly between different tools. As such, the choice of SAST tool is crucial to ensure proper coverage of detectable vulnerabilities. However, developers seem to choose SAST tools based on factors such as reputation and cost, rather than performance [5]. One reason for this could be the distrust that has been observed for SAST tool benchmarks; they are perceived as either being too basic to model real problems or biased in favor of specific SAST tools [5]. Despite this, most previous work comparing SAST tools evaluate using the very thing developers distrust: vulnerability benchmarks. This highlights the need for reliable and unbiased SAST tool comparisons, that more accurately reflect real-world vulnerabilities.

A promising development in vulnerability detection comes from large language models (LLMs). Recent research has begun exploring their ability to identify security vulnerabilities in source code, much like SAST. Unlike SAST tools, LLMs have potential to generalize across programming languages and vulnerability types. To provide vulnerability data for such data-driven vulnerability research, CVEfixes

was created [34]. CVEfixes is a vulnerability database constructed from code patches of real world vulnerabilities in open source projects, as opposed to the synthetic benchmarks that are distrusted by developers.

No previous work was found that comparatively evaluates SAST tools on CVEfixes. Additionally, there is a lack of studies comparing the performance of Java SAST tools and the performance between free-to-use and proprietary SAST tools. Thus, conducting such a comparison could prove valuable to the field and could provide a baseline for future LLM vulnerability detection research.

This thesis aims to comparatively **evaluate the vulnerability detection efficacy, time consumption and vulnerability classification quality of Java SAST tools**, in the interests of Sinch Sweden AB, where this thesis was conducted. The end goal is to determine **which Java SAST tool should be recommended**. Additionally, the results of the comparative evaluation aims to shine a light on **what limitations there are to SAST in terms of vulnerability detection**.

1.1 Disposition

The subsequent chapters present the following content:

- **Chapter 2: Background** introduces concepts within application security testing and relevant previous research within the field. Ways of automating and evaluating testing are introduced.
- **Chapter 3: Tool Selection** covers what tools were evaluated and how the selection process was conducted.
- **Chapter 4: Data set - CVEfixes** contains information about the CVEfixes data set of how it is structured and utilized in this study.
- **Chapter 5: Evaluation Methodology** describes how tests were conducted on the CVEfixes data set, in order to evaluate the tools. How the evaluation results were processed to be presentable is also described.
- **Chapter 6: Results** presents the results from the evaluation in various ways to make it easier to interpret. It specially looks at vulnerability detection rates, execution times and vulnerability classifications.
- **Chapter 7: Discussion** examines the performance of the tools across all the evaluated metrics. Based on these results, the most effective tool is identified and potential limitations of the study are discussed.
- **Chapter 8: Conclusion** summarizes the key findings of the study and highlights notable insights. Additionally, it outlines potential directions for future research based on the results.

Chapter 2
Background

Security testing is widely recognized as an essential part of software development. While the methods used can vary based on preference, the goal remains the same: ensuring the code is secure and free from bugs.

Since mistakes are inevitable, some form of debugging is often needed. This is usually done manually, but automated tests can help reduce the time spent debugging and catch issues early. Automated tests also warn the developer if future changes might break existing functionality. Writing good tests requires a solid understanding of the code and its expected behavior. However, thinking of every possible edge case can be a challenge.

The following section discusses *application security testing*. It focuses on *static code analysis tools* as this is the core of the thesis. The subsequent sections introduce how vulnerabilities from the security testing tools are reported and how the tools can be evaluated. This includes how previous work performed evaluations on such tools and what data sets they have been tested on.

2.1 Application Security Testing

There are multiple approaches to ensuring the security of an application, each offering different strengths depending on the context in which they are applied. According to Bennett et al. [9], manual code reviews remain the most widely used technique, even with the growing availability of automated tools. In their study, the most commonly employed methods beyond manual review, ranked from most to least used, are unit testing, penetration testing, *static application security testing* (SAST), *dynamic application security testing* (DAST) and *interactive application security testing* (IAST).

Unit testing is a classic method in java, where developers write tests for individual components or functions rather than the application as a whole [39]. Typically, the main purpose of unit testing is to ensure correctness and prevent accidental introductions of unwanted behavior to a program.

Penetration testing, by contrast, simulates real-world cyberattacks on systems, networks or applications. Its goal is to uncover and patch vulnerabilities before they can be targeted by malicious actors [21].

Static application security testing (SAST) is a white-box method that scans source code or binaries without running the program. It helps developers catch

potential issues early by identifying insecure code during development [53].

Dynamic application security testing (DAST), on the other hand, operates from a black-box perspective, examining a running application from the outside. It simulates an attacker's behavior by interacting with the application during execution to identify security weaknesses, all without needing access to the source code [53].

Interactive application security testing (IAST) seeks to combine the strengths of both SAST and DAST. It runs within the application during runtime, monitoring internal behavior and data flow to detect vulnerabilities that are not only present in the code but also actually reachable and exploitable during execution [53]. In this thesis, the focus will be on static code analysis.

2.1.1 Static Code Analysis and SAST

Static code analysis (SCA) is the act of examining source code without executing it. This is particularly useful for identifying potential errors, poor code quality or coding standard violations early in the development cycle. *Static Application Security Testing* (SAST) applies static code analysis by scanning for patterns that indicate weaknesses that could result in vulnerabilities [20]. SAST tools typically implement this approach using predefined *rules*, each designed to detect a specific type of security weakness. The result of a SAST tool scan is a list of findings that specify where in the code a rule was matched.

Tests conducted by Elder et al. [20] suggest that SAST tools are able to find more vulnerabilities than other vulnerability testing techniques. Additionally, two popular SAST tools reported that their tools run in linear time. [26, 71]. This would suggest that SAST tools are viable for both smaller and larger projects.

Implementations and techniques

How different SAST tools are implemented varies, but are all founded on the principle of pattern matching. Some tools tokenize the code and turns it into an *abstract syntax tree* (AST) before analyzing it [1, p. 41] [41]. An AST removes the syntax of the language but keeps the logic intact. The simplest implementation of SAST generates one AST per file. A more advanced implementation generates one AST for multiple files, so called *cross-file analysis* [70]. Another important principle in static code analysis is *control flow*. It describes what paths can be taken in the code based on input and represents that data in a graph [1, p. 399]. A very similar concept is *data flow analysis*. While control flow analysis focuses on the execution paths a program may take, data flow analysis is concerned with how data moves and changes along those paths. It tracks variable definitions, uses and how values propagate through a program Aho et al. [1], p. 597[6].

Taint analysis is one of the most widely used principles in SAST, with roots in data flow analysis [2, 41]. It works by tracking how potentially malicious input (tainted data) flows through a program to determine whether it can reach sensitive operations. The point in the code where untrusted input enters the system is referred to as a *source*, while the location where this input may cause harm is known as a *sink*. Any data influenced by this untrusted input is considered *tainted*.

If the data is processed to remove or neutralize the threat before reaching a sink, it is said to have passed through a *sanitizer*. A common example is SQL injection: the point where a user can inject a query serves as the source, a prepared statement acts as the sanitizer and query execution functions as the sink [28].

2.2 Vulnerability Documentation and Classification

CVE (Common Vulnerabilities and Exposures), maintained by the MITRE Corporation, serves as the de facto international standard for identifying vulnerabilities [55]. After a vulnerability is reported to a *CVE Program Partner*, such as a popular software vendor, it is assigned a unique identifier (*CVE ID*). Later, when the stakeholders are ready to publicly disclose the vulnerability, a *CVE Record* is published in the *CVE List*, stating which software versions are affected, along with a CVE ID, a brief description and a list of external references [56]. The CVE List is publicly accessible on their website and a free-to-use API is provided as well.

The *National Vulnerability Database* (NVD), created by the National Institute of Standards and Technology (NIST), builds upon the data provided by CVE by providing additional information on each CVE Record [35, 38]. NVD and CVE are together the most commonly adapted vulnerability databases in security research [38]. Notably, NVD maps each vulnerability to a CWE (Common Weakness Enumeration) identifier, as well as providing impact metrics and other relevant metadata such as suggested solution methods [35].

CWE (Common Weakness Enumeration) is a community-developed system run by the MITRE Corporation that categorizes common software and hardware weaknesses [17]. CWE describe a weakness as "a condition in a software, firmware, hardware or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities" [17]. Perhaps unsurprisingly, many SAST tools associate its detection rules with CWE weaknesses, as its rules match source code with issues that could lead to vulnerabilities of a specific category.

There is no default hierarchy of CWE weaknesses. Instead, there are multiple *CWE Views* that provide means of navigating a hierarchical representation of the CWE List, with varying scope and purpose. There are four types of CWE weaknesses that correspond to varying levels of abstraction in a View: Pillar, Class, Base and Variant weaknesses, with Pillar being the most abstract (see fig. 2.1) [81]. A fifth CWE type, Category, is simply a collection of weaknesses that share some common characteristic, but is not part of a CWE View hierarchy [81].

CWE provides guidelines for how CVEs are mapped to weaknesses. Generally, mapping of CVEs to higher-abstraction CWE weakness types is either prohibited or discouraged. Specifically, mapping of CVEs to CWE Categories is always prohibited [81]. Because of an ever changing cybersecurity landscape, certain CWE weaknesses become less relevant over time and are marked deprecated. Mapping to deprecated weaknesses is also prohibited.

Two interesting CWE Views include *CWE Top 25* [85]: A list of the top 25 most dangerous weaknesses; and *OWASP Top Ten* [84]: A list of CWE Classes correlating with OWASP Top 10. Another View, *CWE-1000: Research Concepts* [83], is stated as facilitating research into weakness types and organizes weaknesses

by behavior in all levels of abstraction. There are a total of 10 Pillar weaknesses in CWE-1000, including for example *Improper Access Control* and *Improper Neutralization* [83]. In contrast to most other views, the intent is for CWE-1000 to encompass *all* CWE weaknesses [83].

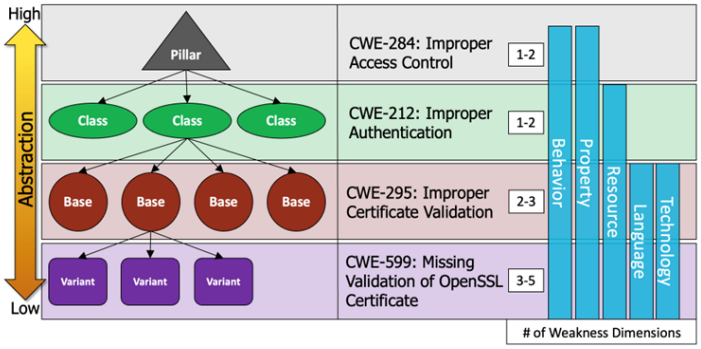


Figure 2.1: Abstraction levels of CWE weaknesses[1].

2.3 Previous work

A review of previous work was conducted to identify prior work relevant to vulnerability detection and SAST tool evaluation. The review focused on three areas: comparative evaluations of Java SAST tools, vulnerability data sets and methods for vulnerability identification and classification. Literature was sourced primarily from peer-reviewed articles and official documentation. Initial searches using LUBsearch, Lund University's internal scientific database, yielded few relevant results. Subsequently, Google Scholar was used as the primary repository for scientific articles, while Google Search was used to find other documentation. A few additional articles were identified through references in found articles.

The review findings are summarized in the following two subsections. Section 2.3.1 non-exhaustively lists and describes vulnerability data sets used in previous work. Section 2.3.2 describes previous approaches to comparative evaluations of SAST tools.

2.3.1 Vulnerability Data Sets

Until now, previous work [3, 34] has mainly evaluated SAST tools on synthetic data sets such as *the Juliet Test Suite*, created by the *National Security Agency* (NSA) [80]. While Juliet covers a wide range of weaknesses and vulnerabilities, concerns have been raised of its use in evaluating vulnerability detection [10, 11]. Researchers state that Juliet's vulnerabilities are not very diverse and that many of

[1] CWE is sponsored by the U.S. Department of Homeland Security (DHS) Cybersecurity and Infrastructure Security Agency (CISA) and managed by the Homeland Security Systems Engineering and Development Institute (HSSEDI) which is operated by The MITRE Corporation (MITRE). Copyright © 2006–2025, The MITRE Corporation. CWE, CWSS, CWRAF, and the CWE logo are trademarks of The MITRE Corporation. https://cwe.mitre.org/about/termsofuse.html

them are of synthetic nature that would not occur in real-world software [11]. Similar concerns for the use of synthetic data sets in evaluating SAST tools has been raised by other researchers [10]. Additionally, the Juliet Test Suite is outdated, with the latest version 1.3 last released in 2017 [80].

By collecting examples of real vulnerability fixes from open source projects, a more representative data set can be created. The National Vulnerability Database (NVD) publicly provides vulnerability feeds of updates to CVE records. Each time a reported vulnerability is fixed in an open source project, its CVE record will be updated with references to the source code repository and the vulnerability fixing commit [11].

CVEfixes, introduced by Bhandari et al. [11], automatically collects and complements data from these NVD vulnerability feeds into a comprehensive vulnerability data set containing the source code before and after the vulnerability fix. This is done in three layers of abstraction: commits, file changes and method changes. In addition, many metadata fields are provided, such as the programming language of each modified file. In contrast to the Juliet Test Suite, CVEfixes is regularly updated. In addition, the code to reproduce CVEfixes is open-source, on an MIT license. Hence, the database can be recreated at any time to include the most recent vulnerability fixes.

Magma, introduced by Hazimeh et al. [31] and used by Lipp et al. [40], is another notable vulnerability data set sourced from a handful of open source C/C++ projects. Its purpose is to provide ground-truth for evaluating and comparing *fuzzers*. Fuzzers procedurally generate inputs to a program with the purpose of triggering a fault and identifying the corresponding bug. Magma notably labels both a bug's root cause and its manifestation, determined by manual inspection. The concepts of root cause and manifestation are highly related to sources and sinks in taint analysis. The root cause is risky code that alone may not cause a bug, while the manifestation arises from the misuse of the risky code, creating a security issue [40]. For example, an integer overflow (root cause) can lead to an out-of-bounds write (manifestation) when used improperly. While created with fuzzers in mind, Magma is equally applicable to SAST evaluation, as seen by Lipp et al. [40].

For Java SAST evaluation of real-world vulnerabilities specifically, there are two data sets in particular worthy of mention, other than CVEfixes: the data sets introduced by Ponta et al. [54] and Li et al. [37].

Ponta et al. [54], curated a data set of vulnerabilities sourced from open source Java projects, in the interests of the German software company SAP. The data was sourced partly from NVD and partly from project-specific web resources, collecting the vulnerability-fixing commits of 624 publicly disclosed vulnerabilities from 205 projects. Only projects that had practical industrial relevance were included, specifically the ones used in SAP products or internal tools. Not all vulnerabilities included had a CVE identifier or an NVD entry. The data set by Ponta et al. [54] is provided as an open source CSV file, with the following column information: vulnerability ID, repository url, commit ID and vulnerable label.

Li et al. [37] construct their own Java vulnerability data set for a comparative SAST tool evaluation. The data is sourced from NVD, Debian and Red Hat Bugzilla and is manually curated to only include projects that are *package-*

able. This was required for them to evaluate SAST tools that analyze bytecode as opposed to source code. In total they collected 165 package-able open source programs containing 165 unique CVEs.

CVEfixes was chosen as ground-truth for the evaluation, in the hopes of it being representative of vulnerabilities occurring in real software and because it provides useful abstractions complementing the data from NVD. Unlike Li et al. [37], this thesis evaluates SAST performance on uncompiled source code. This approach eliminates the need for manual review and allows for the direct and automated use of the CVEfixes dataset. The process for regenerating CVEfixes, including most recently disclosed vulnerabilities, is fully open source. This ensures reproducibility and makes the dataset well-suited for future evaluations.

2.3.2 Evaluating SAST tools on Real-world Vulnerabilities

While there are numerous previous works evaluating SAST tools on synthetic vulnerability benchmarks [3, 4, 19, 22, 30, 34, 49, 51], using real-world vulnerabilities as the ground-truth is a less common evaluation method, despite its potential.

The idea of evaluating static code analysis tools on bugs from real-world projects is not new. In 2012, Thung et al. [86] constructed a data set from Bugzilla and JIRA bug reports for three open source projects and cross-referenced them with their corresponding version control revisions (both CVS and Git). By manual inspection, they then determined the faulty lines of code that contained the root cause of the bug. The authors then evaluated each chosen static code analysis tool by the amount of overlap between tool-flagged and the manually determined faulty lines of code.

Lipp et al. [40] built upon the work of Thung et al. [86] by applying a similar method for comparatively evaluating SAST tools on C programs. For the evaluation data set, they supplement the Magma vulnerability data set [31], described in section 2.3.1, with well-documented vulnerabilities from two additional open source projects. Instead of manually identifying specific lines of code containing a bug's root cause, like Thung et al. [86], the authors introduced function-level detection. Here, a vulnerability is deemed detected if the static analyzer flags any function modified in the vulnerability-fixing patch. As noted by the authors, static analyzers typically flag the manifestation of a bug as vulnerable, rather than its root cause to limit false positives. In contrast, code patches typically address the root cause [40]. Assuming this distinction holds for a given vulnerability, if the root cause and manifestation are not contained within the same function, determining whether a tool correctly detects the vulnerability is impossible without manual inspection, since the tool may only flag the manifestation. The authors validate their approach of using function-level detection by analyzing the Magma data set, concluding that for 92% of vulnerable C functions, the root cause and manifestation are contained within the same function. Finally, the authors evaluated not only whether a vulnerability flagged the correct vulnerable file, but also considered the correctness of its CWE classification. The static analysis tools were evaluated both with and without considering the correctness of CWE labeling.

Li et al. [37] perform a comprehensive comparison of the vulnerability detection of a broad range of free-to-use SAST tools on Java code. By following

Tools	S_{F-C}					S_{M-C}				
	$\#D_{\text{vul}}$	$\#FP$	$PP_{\%}$	$TP_{\%}$	Pr	$\#D_{\text{vul}}$	$\#FP$	$PP_{\%}$	$TP_{\%}$	Pr
CodeQL	15	9	9.0%	3.6%	40%	11	5	6.7%	3.6%	55%
Contrast	2	2	1.2%	0%	0%	1	1	0.6%	0%	0%
Horusec	38	23	23%	9.0%	39%	21	6	13%	9.0%	71%
Insider	11	11	6.7%	0%	0%	4	4	2.4%	0%	0%
SBwSFB	26	25	16%	0.6%	3.8%	17	1	10%	0.6%	5.9%
Semgrep	31	26	19%	3.0%	16%	9	4	5.5%	3.0%	56%
SonarQube	12	8	7.3%	2.4%	33%	9	5	5.5%	2.4%	44%

Table 2.1: The number of detected vulnerabilities and approximate false positives from the results of Li et al. [37].

a well-motivated tool selection process, seven SAST tools were chosen for evaluation: CodeQL, Contrast, Horusec, Insider, SpotBugs with Find Security Bugs, Semgrep and SonarQube. The authors evaluated the SAST tools both on the OWASP benchmark [25], consisting of synthetic vulnerabilities, and on their own Java CVE data set, as described in section 2.3.1, which collects historical data from open source projects. The authors applied both function-level and file-level detection, where the latter considers a vulnerability detected if any of the files modified in the vulnerability-fixing patch are flagged as vulnerable by the SAST tool. Like Lipp et al. [40], evaluation was performed both with and without accounting for correct CWE labeling, in different detection scenarios. The number of detected vulnerabilities, as well as an approximate number of false positives (See section 2.4) are both presented for two detection scenarios: S_{F-C} and S_{M-C}, as defined in section 5.3.1. These results are presented in table 2.1. $\#D_{\text{vul}}$ represents the number of detected vulnerabilities and $\#(D_{\text{vul}} \cap D_{\text{patch}})$ is the number of approximate false positives.

A more novel approach to vulnerability detection is to leverage machine learning and in particular large language models (LLM). A systematic literature review conducted by Zhou et al. [88] concludes that current LLM-based vulnerability detection solutions generally have small-grained input granularity, meaning they target smaller code abstractions like functions or lines of code. One key reason for this is stated as being the limited input length of the lightweight LLMs predominantly used in research, which align well with function-level detection but struggle to scale to repository-level detection. As opportunities for future work, the authors highlight the need for both research in repo-level detection, as well as research in developing LLMs customized for vulnerability detection.

In a following study, Zhou et al. [87] address both these research opportunities by performing a direct comparison between vulnerability detection of SAST tools with fine-tuned LLMs employing repo-level detection. The authors argue that since SAST tools are used to detect vulnerabilities across entire code repositories, applying LLMs to the same detection scope makes for a more fair comparison. The authors compared the vulnerability detection for three programming languages, including Java. Repo-level scanning was performed by running the LLMs with one function at a time as input for an entire code repository and then aggregating the results, flagging each function as either vulnerable or non-vulnerable. The SAST evaluation followed Li et al. [37] with the exception of any CWE classification. Their results show LLMs outperforming SAST tools in terms of accuracy, but with significant false positive rates. The best performing LLM for Java vulnera-

bilities flagged 22% of functions, whereas the true vulnerable function ratio ranged between 0.008% to 0.18% in the data set.

While Zhou et al. [87] technically apply LLM vulnerability detection to repo-level detection, the detection capabilities are still limited by input length. In addition, without accounting for vulnerability classification, the accuracy of such models cannot be trusted as the sole metric, as any suspicious-looking function could be marked vulnerable, without there being any way of verifying the models' understanding. This thesis instead accounts for CWE labeling, only considering a vulnerability detected if it can be correctly classified in its weakness class. Additionally, the input of the SAST tools are limited by only containing a file or a set of changed files at a time, comparable with lightweight LLMs' limited input length. This provides a baseline that can be used to evaluate future vulnerability classification machine learning models.

2.4 Metrics

Bennett et al. [10] identify an important benefit with evaluating on synthetic data sets: There is high confidence that no other unlabeled vulnerabilities exist in the data set, meaning any other vulnerabilities detected by the SAST tools can safely be considered false positives. As such, the key metric precision and in extension, precision-reliant metrics such as F1 score, can safely be used without affecting the validity of results. Precision-reliant metrics are used in a number of papers evaluating SAST tools on synthetic data [3, 4, 22, 48] .

In contrast to synthetic data set evaluation, data sets gathered automatically from real-world vulnerabilities have a less reliable ground truth. Historical data from real-world code repositories may contain undiscovered vulnerabilities [88], meaning it is not possible to label false positives without labor intensive manual inspection. This makes precision and in extension, the F1 score metric, unreliable. As Bennett et al. [10] note, this makes synthetic data sets tempting to use for evaluation, but results will likely not be representative of performance in production settings.

In many previous studies evaluating SAST efficacy on real-world vulnerabilities, the primary metric used is either the number of vulnerabilities each tool detects or the percentage of total known vulnerabilities it is able to detect [12, 37, 40, 88]. Zhou et al. [88], Lipp et al. [40] and Li et al. [37] all employ some form of metric approximating the rate of false positives. The first two papers [40, 88] approximate false positives by calculating the marked function ratio:

$$\frac{\#\text{Flagged Functions}}{\#\text{All Functions in Benchmark}} \tag{2.1}$$

The motivation behind this metric is that only a small fraction of functions are vulnerable in the repositories, meaning that the majority of flagged functions are non-vulnerable [88], i.e. false positives.

Li et al. [37] introduces a new metric for the same purpose: the proportion of vulnerabilities marked as vulnerable both before and after a vulnerability-fixing commit:

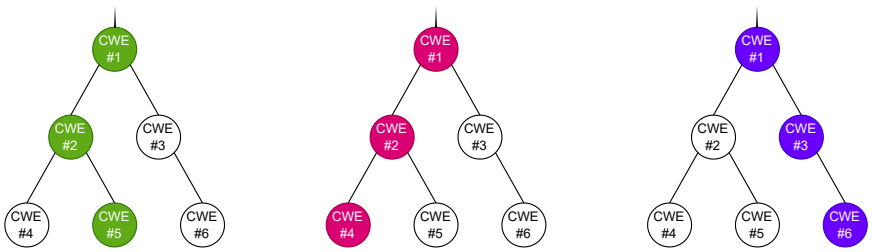
$$\frac{\#(D_{\text{vul}} \cap D_{\text{patch}})}{\#D_{\text{vul}}} \tag{2.2}$$

Here, D_{vul} and D_{patch} represent the set of detected vulnerabilities in the vulnerable and patched versions, respectively [37]. The # denotes the size of a set.

Determining if a vulnerability is correctly classified is not necessarily a binary question. From the CWE-1000 view, some weaknesses are more closely related than others. An incorrectly classified, but closely related vulnerability could be considered more correct than a distant one. Pan et al. [52] tried to encompass this idea by providing a score between 0 and 1 rather than exclusively 0 or 1, naming it the Path Fraction (PF) score. The metric builds upon the fact that CWE-1000 structures the relations between different classifications as a tree. The more an incorrect classification follows the same path down the "CWE tree" as the correct classification, the higher the score. The PF score is the average of a data set, defined as:

$$\frac{1}{N}\sum_{j=1}^{N}\frac{\#(\hat{Y}_j \cap Y_j)}{\#Y_j} \tag{2.3}$$

Y symbolizes the true path of the a vulnerability, $\hat{Y}$ the classified one and N, the number entries in the data set. Here, the # marks the number of nodes in the path. In fig. 2.2, an example of the PF score is calculated is presented. Here, classification A (2.2b) shares more nodes with the ground-truth path (2.2a) and would therefore receive a better PF score than classification B (2.2c).



(a) Truth **(b)** Classification A **(c)** Classification B

Figure 2.2: Example of how the PF score is calculated.

This metric does not take into consideration how many siblings a classified node has. If a CWE node in the tree has many children, guessing which of its children is on the correct path is less likely than if the node has few children. Accounting for this could potentially produce a more fair metric.

Chapter 3
Tool Selection

Considering the large number of available SAST tools, selection criteria were developed to select a small set of SAST tools to include in the comparison. Inspiration was drawn from Li et al. [37], with the addition of proprietary tools and source code compatibility. Instead of focusing only on free and open source tools, an effort was made to also include commercial tools, as long as they adhere to the remaining criteria. This change was made for two reasons: For the comparison to be as useful as possible for Sinch and other software businesses; and for the comparison to better represent the cutting-edge of SAST. Additionally, the tools were required to be *source code compatible*, as defined in the selection criteria below:

- **Java support**: To limit the scope of the thesis; only tools that support Java were included.
- **Actively maintained**: The comparison was limited to tools that are actively maintained, making the results more useful for future use in development. Any tools that had not been updated in 2 years were excluded.
- **Command-line interface**: Utilizing a command-line interface (CLI) is practically essential for the automated evaluation of each tool. As such, any tool lacking CLI support were excluded.
- **Security-related**: Only security-related static code analyzers were included, assuming non-security-related tools lack in performance.
- **CWE Reporting**: Only tools with well-documented rules with associated CWE weaknesses were included in order to evaluate vulnerability classification quality.
- **Source-code compatible**: Because of time constraints, the code repositories could not be compiled before scanning. Each code repository has unique build configuration and collecting buildable repositories would be prohibitively time consuming. As such, any tool requiring the code to be compiled were excluded.

As these criteria still encompass a lot of tools, open-source tools were also selected by descending number of Github stars. The following tools were taken into consideration:

Tool	Latest Update[1]	CLI	Security focused	CWE format	Github stars[1]	Free version available	Source-code compatible
PMD [59]	29/11/24	✓	×	×	4.9k	✓	✓
FSB [58]	26/2/24	✓	✓	✓	2.3k	✓	×
Infer [23]	21/6/24	✓	×	×	15k	✓	✓
SonarQube [79]	2/12/24	✓	✓	✓	9.2k	✓	✓[2]
CodeQL [27]	9/12/24	✓	✓	✓	7.8k	✓	✓
Fortify tiny [18]	31/10/24 [24]	✓	✓	✓	-	×	✓
Snyk [76]	12/12/24	✓	✓	✓	5k	×	✓
Semgrep [60]	18/12/24	✓	✓	✓	10.8k	✓	✓
Bearer [57]	22/11/24	✓	✓	✓	2.1k	✓	✓
Horusec [33]	8/6/22	✓	✓	✓	1.2k	✓	✓
Insider [63]	26/1/21	✓	✓	?[3]	519	✓	?[3]
CheckMarx [13]	6/10/24 [14]	✓	✓	✓	-	×	✓
Contrast [62]	29/8/24 [61]	✓	✓	✓	-	×	×

Table 3.1: Java SAST tools considered for evaluation, and their compliance with the selection criteria.

The considered tools were selected based on what had been used in previous papers [22, 37, 87] and from the `static-analysis` topic on Github [29]. A cutoff for the number of Github stars was arbitrarily chosen at 500.

Based on the information from Table 3.1, some of the tools did not meet the criteria. Although both PMD and Facebook Infer are SCA tools, they aren't specifically tailored for security and where therefore excluded. Horusec and Insider haven't been updated recently whereof Insider's website is no longer in operation; the project might have been abandoned. Though Horusec hasn't been updated in two years, results by Zhou et al. [87] suggest that its performance is still comparable to other tools. As such, it was still included in the evaluation. Most of the tools support source code scanning, with SonarQube offering partial support. It can only scan uncompiled Java code if a single Java file is present. Although its support is partial, it was included. Out of the 13 tools evaluated, 4 have no free options and require licensing.

The ultimately chosen tools can be seen in table 3.2. The table also includes some of the analysis types the tools have implemented.

[1] As of 18/12/2024
[2] Partially compatible
[3] Information unavailable

Tool	Edition	Analysis type				Sources
		Cross-file	Control flow	Data flow	Taint	
Bearer	Free	? [4]	✓	✓	✓	[8]
CodeQL	Free	✓	✓	✓	✓	[16, 15]
Horusec	Free	×	×	×	×	[32, 37]
Semgrep	Pro	✓	✓	✓	✓	[70, 67, 73, 72]
SonarQube	Enterprise	✓	✓	✓	✓	[64, 74, 78, 77]

Table 3.2: Chosen SAST tools and some of their implemented analysis types.

3.1 Licensing

Premium licenses for SonarQube and Snyk were acquired. Due to Snyk's usage limitations, the license was not sufficient to achieve adequate results and was therefore excluded. Through mailing contact, none of the other companies provided licenses for academic purposes.

[4] Unclear information

Chapter 4
Data set - CVEfixes

The use of CVEfixes as the evaluation data set was decided early on in the planning phase as per recommendation from the project's supervisor. Upon initial review, it was considered suitable for the evaluation for its collection of real-world vulnerabilities, convenient structure and reproducibility. Additionally, no other studies were found that evaluate SAST tools on CVEfixes. As such, CVEfixes was chosen and no further rigorous selection process was conducted.

CVEfixes is a data set containing data about vulnerable code, collected from real Git repositories. It is periodically updated to comply with the CVE classification and to include newly disclosed vulnerabilities [11]. The collection process of vulnerabilities is automated by extracting information from NVD as well as code from their corresponding Git repositories. Since the CVEfixes project is open source, it is possible to generate a newer version manually if more up-to-date information is needed than what the latest release provides. CVEfixes version 1.0.8 [47] was used during the tool evaluation, which was the latest release at the time. The data is contained within an SQL database.

The following sections describe how the database is structured, what parts of it are useful and how it can be filtered to suit the needs of the project. The last section touches on limitations of CVEfixes.

4.1 Database structure

The data set is divided into 8 SQL tables that can be split into two sections: classification and code (see fig. 4.1). The four tables `cve`, `cwe`, `cwe_classification` and `fixes` belong to the classification category and contains information about CVEs, CWEs and how they are connected. `cve` is the largest one, consisting mainly of vulnerability scores of different metrics. `cwe` has a couple of fields whereof one links to the official documentation with more information. `cwe_classification` and `fixes` simply serve as bridges between the `cve` and `cwe` tables and `cve` and `commits` tables respectively. The remaining four tables, namely `repository`, `commits`, `file_change` and `method_change` belong to the code section and describe different abstraction layers of a Git project [11]. The highest abstraction layer, the `repository` table, most importantly, contains the field `repo_url` and descriptive fields of less importance to this project. The second highest layer, the `commit` table, similarly contains a lot of non-essentials, beside from the `hash` field. This field

corresponds to the hash of the Git commit that fixed a vulnerability. The second lowest abstraction layer, the `file_change` table, contains information about the files modified in a commit. Each commit can have multiple corresponding entries in the table. Each entry describes the file both before and after the commit with appropriately named keys for differentiation. The field `change_type` can be used to determine if a file has been `modified`, `deleted`, `added` or `renamed`. In the cases of `deleted` and `added`, fields corresponding to after and before respectively are not present. The last and lowest layer, the `method_change` table, goes one step further to describe how methods have been changed. Each entry in `file_change` can have many corresponding entries in `method_change`. In contrast to `file_change`, `method_change` does not contain information on both before and after a fix; the information is divided into two entries. The boolean field `before_change` determines if the entry is from before or after the commit.

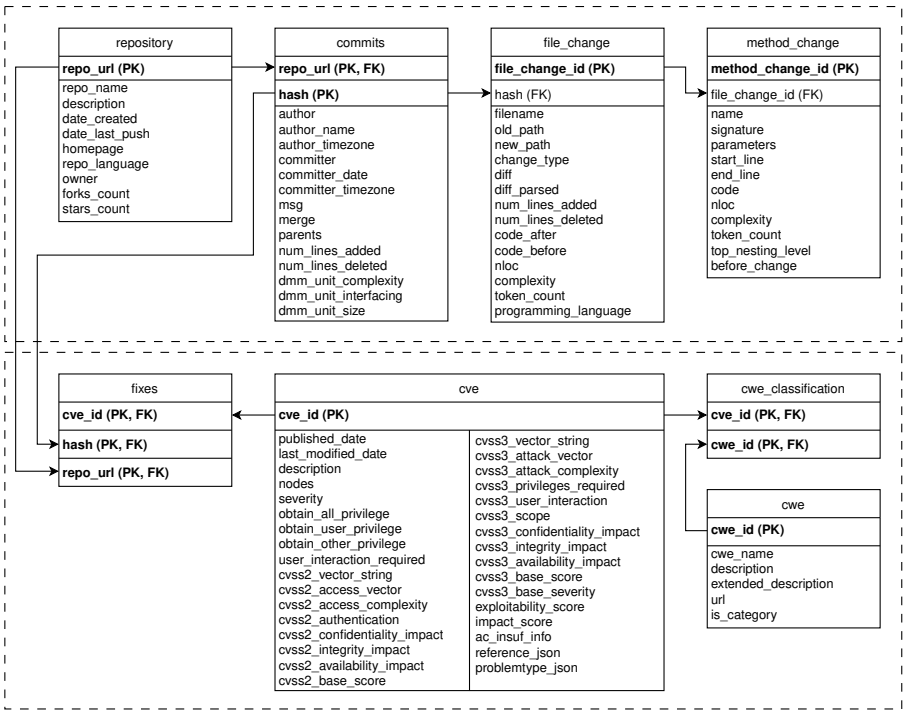


Figure 4.1: Entity Relation Diagram of the CVEfixes data set. The diagram is structured into two sections: a code section (upper) and a classification section (lower).

4.2 Exploratory Analysis

An initial exploratory analysis was performed on the data set in order to identify any limitations that could affect the practical implementation of the SAST tool evaluation, and to determine what data needed to be filtered out. To perform the

exploratory analysis, the CVEfixes database was queried using a number of SQL statements, of which the resulting data was processed with Python scripts. The exploratory analysis was unstructured and guided by insights from the concurrent literature study.

In order to adhere to the project constraints, it had to be determined how the data set could be filtered to contain only Java vulnerabilities. The goal was to scan not only Java source files, but also other files relevant to Java, such as configuration files. For each file change, CVEfixes provides a column `programming_language` that supposedly contains what programming language the file is written in. In order to ensure reliability of its data, the file distribution for Java files was explored. It was found that of files that were identified as Java, roughly 6% were without a `.java` extension. Instead, everything from C source files to TypeScript files were mislabeled as Java. Similarly, of files with the `.java` extension, 2% were not labeled as Java. Both of these inconsistencies were verified by manual inspection. From these findings, a decision was made to not rely on the `programming_language` column provided by CVEfixes and instead trust file extensions to identify files relevant to Java development, especially Java source files.

To determine whether special handling was needed for missing or placeholder values, the distinct values of each column relevant to the evaluation were investigated. The first finding was that the associated CWE weaknesses of some vulnerabilities were shown to contain placeholder values, namely `NVD-CWE-noinfo` and `NVD-CWE-Other`. These labels are used to indicate lacking information from CVE and lack of a suitable CWE mapping, respectively. Any vulnerabilities associated with such placeholder CWEs were excluded from the evaluation, since there is no ground truth for its vulnerability classification. Another finding was that `code_before` and `code_after` contained many placeholder values `None`, which initially caused some concern. Through further investigation, it was shown that the placeholder value only appeared for file creation and deletion. Finally, `file_change.nloc` was shown to include many `nan` values, for reasons still unknown. Given that the source code of the files were provided, the number of lines of code could easily be calculated. Other than that, no unexpected empty or placeholder values were found in data affecting the evaluation.

Two other issues were identified related to CWE mapping. Firstly, some code changes referenced deprecated CWE issues, to which CWE mapping is prohibited. Secondly, some vulnerabilities in the data set were mapped to CWEs not within CWE-1000. They were shown to be CWE Categories, to which mapping is also prohibited.

To ensure the reliability of the comparative analysis, the data was checked for duplicates. In particular, hash collisions were considered, as the data set includes many repositories, including forks. Duplicate hashes were found and in all cases, collisions occurred because of the same code changes existing in different forks of the same code repository.

Category	File Extensions	Usage
Source Code	`java`	Java source code.
Build & Dependency	`bazel`, `gradle`, `kts`, `properties`, `mf`	Build scripts, dependency management, metadata for builds and JARs.
Application Configuration	`conf`, `yaml`, `yml`, `xml`, `json`, `toml`, `ini`	Defining application settings or framework configurations.
Security & Certificates	`crt`, `key`	SSL/TLS certificates and keys for encryption.
IDE & Project Metadata	`classpath`, `iml`, `.gitignore`	IDE project settings and version control exclusions.
Template Engines & Views	`ftl`, `ftlh`, `vm`, `jelly`, `jsp`	Templates for generating dynamic web content.
Framework-Specific	`aj`, `thrift`	Files for AspectJ (AOP) and Apache Thrift (RPC framework).

Table 4.1: File extensions included in the scans and their respective uses in Java development.

4.3 Data Pre-processing

Before conducting the experiments, the data provided by CVEfixes was filtered. As discussed in the previous section, the `programming_language` columns could not be relied on to filter for Java files. Instead, a choice was made to only include code patches modifying at least one Java source file. These commits were shown to include source files of multiple other languages. To focus only on files relevant to Java development, only the file types listed in Table 4.1 were included in the evaluation.

These file extensions were selected from an exhaustive list of file extensions present in the code patches with at least one modified Java source file. These include, for example, source files, configuration files, certificates and templates. The reason why only these file extensions were included is to avoid scanning source files for languages other than Java, as this would produce inaccurate results. At the same time, files relevant to Java development were included to avoid restricting the scope of detectable vulnerabilities.

To prevent overrepresentation of specific weakness classes, duplicate code patches (with equal commit hashes) were excluded. Additionally, patches only addressing deprecated CWEs, as defined by *CWE-604: Deprecated Entries* [82], were omitted from the evaluation to avoid the scenario of SAST tools mislabeling these vulnerabilities despite correctly detecting them.

Some fixes in CVEfixes were not mapped to any CWE, instead labeled as either `NVD-CWE-noinfo` or `NVD-CWE-Other`. Similarly, some fixes were associated with a CWE weakness or class that was not covered by CWE-1000. In both cases, such code fixes were excluded from the evaluation.

Before filtering, the data set spanned 4249 code repositories, 11873 unique CVEs and 272 unique CWEs. After filtering for non-duplicate Java code changes

associated with non-deprecated CWEs, there were 244 code repositories, 462 unique CVEs and 87 unique CWEs left. The distribution of CWEs into CWE Pillars, according to *CWE-1000: Research Concepts* [83], is illustrated in figure 4.2. There were no code changes in the filtered data set that were associated with *CWE-697: Incorrect Comparison* [45], resulting in 9 CWE Classes left for the evaluation in the CWE-1000 pillar level.

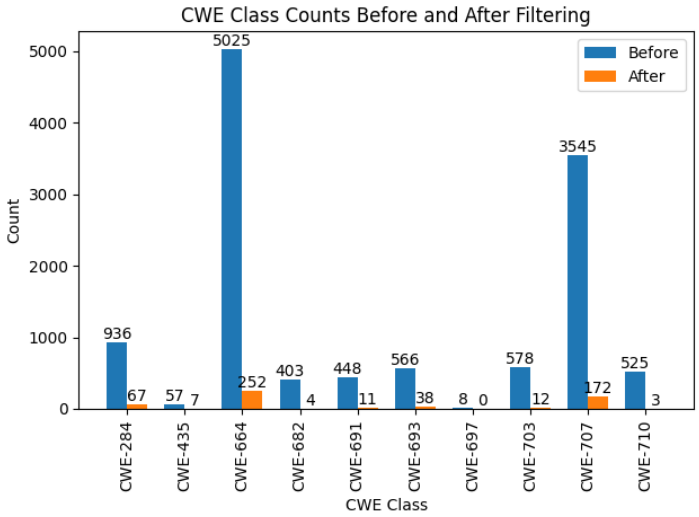


Figure 4.2: CWE Pillar count before and after filtering the data set.

4.4 Data set limitations

As the CVEfixes data set was chosen without a comparison to other data sets, it is important to be aware of possible limitations it might possess. Comparing it to data sets mentioned in previous papers can uncover its shortcomings.

The data sets used by Li et al. [37] and Ponta et al. [54] are manually constructed and curated. This allows the information of each entry to be verified and therefore presumably contain few mistakes. The CVEfixes data set does not have this luxury and may contain inconsistencies arisen from being generated. One such instance already surfaced during the exploratory analysis, namely the unreliable `programming_language` field. Notably, the data set used by Ponta et al. [54] also contains more Java entries than the CVEfixes data set.

Previously, synthetic datasets like the Juliet test suite [80] were considered limited in diversity and unrepresentative of real-world code [11]. However, this allows full control over the code's content, ensuring it contains only what is intended. The code in CVEfixes does not guarantee that it only contains the intended vulnerability and may contain additional vulnerabilities. This makes it difficult to determine if a SAST tool is wrong if it flags an undocumented vulnerability.

Although limited to C/C++, the Magma dataset [31] provides both the source

and sink locations for each vulnerability. This allows a tool to be considered correct whether it flags the source or the sink. In contrast, CVEfixes provides only one of the two, making it impossible to verify correctness if the tool flags the "incorrect" of the two.

Chapter 5
Evaluation Methodology

In order to provide an objective basis for recommendation, a quantitative study was performed, comparing the different Java SAST tools. Each tool was evaluated based on its vulnerability detection efficacy and resource consumption on real-world vulnerable code from open source code repositories, as provided by CVEfixes. A quantitative study was preferred over a qualitative one to focus on the tools' primary purpose in finding bugs, rather than, for instance, qualitatively assessing user experience. Real-world vulnerabilities were chosen as grounds for evaluation in favor of vulnerability benchmarks to be more representative of real use in a production environment.

The following sections describe the evaluation process, including the vulnerability scanning, result processing and result presentation. The chapter concludes with a short discussion of alternative approaches that were considered, but ultimately not pursued.

5.1 Vulnerability Scans

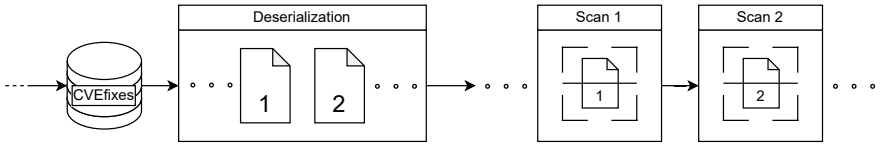
The evaluation was divided into three levels of scanning: file-, commit- and project-level scans. The file level and commit level utilize the `commits` and `file_change` tables of CVEfixes, while the project level scans the entire code repository. Each level consists of two tests: before and after a vulnerability has been fixed. The tools were tested using their respective CLI tools. The following subsections describe how each level was performed except the last one, describing how the tools were configured.

5.1.1 File- and Commit-level Scans

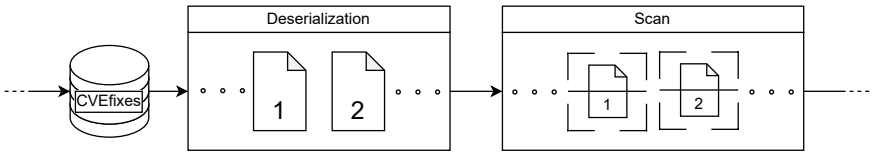
Albeit being divided into two different levels, the file and commit levels are very similar. They both perform scans on all of the files that have been modified in a given commit. The difference is that file level performs one scan per individual file while commit level performs one for all of the files together. This is repeated for both before and after a fixing commit (see fig. 5.1).

All the files that belong to a commit are gathered from the `file_change` table with the corresponding commit hash. The files are put in specific directories based

on the fields `old_path` or `new_path` depending on if a scan is before or after a fix. Correspondingly, the code is fetched from `code_before` or `code_after`.



(a) File level



(b) Commit level

Figure 5.1: Scanning process for the file and commit levels. The scan section is repeated for `code_before` and for `code_after`.

5.1.2 Project-level Scans

In contrast to the other two levels, the code scanned on this level does not come from the data set, but is fetched directly from the corresponding Git repositories (see fig. 5.2). The URL for each repository is retrieved from the `repo_url` field in the `commits` table. Cloning the repositories and checking out to the given commit corresponds to the code after a fix. Checking out to the commits parent naturally yields the code from before the fix. Each project is scanned in its entirety without file-type filtering.

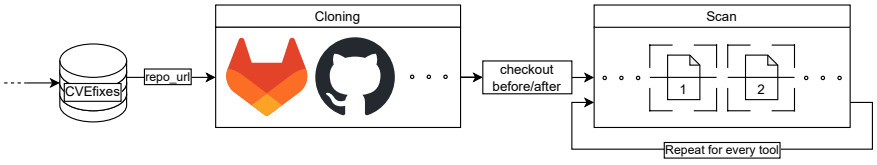


Figure 5.2: Scanning process for the project level. The scan section is repeated for the code before and after the fix [1].

5.1.3 Tool Configuration

To ensure a fair comparison, certain tools were configured with restricted settings, as some SAST tools offer features beyond traditional static analysis. Since scan-

[1] GitHub logo used with permission under the GitHub Logo Policy. © GitHub, Inc.
GITLAB is a trademark of GitLab Inc. in the United States and other countries and regions

ning is time consuming, some features were not used in order to save time. The following subsections describe how each tool was configured.

SonarQube

```
sonar-scanner -Dsonar.projectKey=$PROJECT_KEY
    -Dsonar.sources=$TESTING_FOLDER
    -Dsonar.host.url=$URL -Dsonar.login=$SECRET
```

SonarQube has a lot of functionalities, but for full functionality, it requires the scanned code to be compiled. It does support scanning on Java source code directly, but can only do so if there is a single Java file present. Tests for SonarQube were therefore limited to only file-level scanning. Comparing SonarQube with other tools will therefore also be limited to file level.

The scan results can not be retrieved directly from the CLI tool. Instead, after a scan has been conducted, the results are pushed to a server. The results can be accessed by API calls to the server in between each scan, where only the most recent scan can be accessed by applying a time filter.

CodeQL

```
codeql database create $DB_LOCATION --language=Java
    --source-root=$TESTING_FOLDER --build-mode=none
    --overwrite
codeql database analyze $DB_LOCATION
    --format=sarif-latest --output=$OUTPUT_FILE
```

CodeQL works very differently from the other tools. Before a scan can be conducted, CodeQL builds a code database. By default, the tool tries to build the project, but this can be disabled with the `--build-mode=none` flag. As another default, it tries to download all dependencies if it finds Maven, Gradle or Ant files. Since this feature is not available for direct source code scanning in other tools, it was determined to be unfair. It would also significantly add to the scanning time. As the tool doesn't allow turning this feature off, a workaround is to delete all configuration files of the build tools.

Since every scan requires its own database, it is convenient to recreate the database in the same directory. By default, CodeQL doesn't allow to override a previously generated database. To allow for this, the `--overwrite` flag is required. One database is created for each language, so the `--lang=java` makes sure only one is created. This also prevents the tool from scanning other languages.

Semgrep

```
semgrep scan --config p/ci $TESTING_FOLDER --json
    --json-output=$OUTPUT_FILE --no-git-ignore
```

Semgrep is built upon rule sets. There exists default rules, pro rules and user created rules. In order to use the pro rules, logging in is required. This can be

done by running the command `semgrep login`. Since there are user created rules, Semgrep's accuracy could depend on how many rules are used. Choosing what rules to include requires a lot of subjective decision-making, so choosing among one of Semgrep's own rule sets makes it easier. The `ci` rule set, is claimed to produce few false positives [65]. It contains 111 Java rules (32 default and 79 pro) which is considerably less than the default rule set of 261 rules (118 default and 143 pro) [68]. Using fewer rules should improve the runtime. For these reasons, the `ci` rule set was chosen.

Without any configurations, Semgrep believes that the specified directory is a Git repository. To tell Semgrep that it is not, the flag `--no-git-ignore` can be used.

Bearer

```
bearer scan $TESTING_FOLDER -f=json
   --output=$OUTPUT_FILE
```

Compared to the previous tools, Bearer is very straightforward and requires no special configurations to work as intended.

Horusec

```
horusec start --disable-docker=true -o=json
   -O=$OUTPUT_FILE -p=$TESTING_FOLDER
```

The default behavior of Horusec is to use Docker containers to scan code. To make it more lightweight, `--disable-docker=true` disables this feature.

5.2 Data Post-processing

To prepare the scan results for comparison, a two-step post-processing procedure was applied. This involved two steps: (1) mapping each SAST tool finding to a list of CWE weaknesses and (2) linking each CWE to its position in the CWE-1000 hierarchy.

5.2.1 Mapping Scan Results to a List of CWE Findings

To enable comparison, scan outputs were first normalized to a common format. While there exists an industry-standard for reporting the results of static code analysis tools: *SARIF* (Static Analysis Results Interchange Format) [50], it is not implemented by all tools considered for evaluation. The SARIF format also is not developed specifically for vulnerability reporting and as such has no standardized way to state weaknesses for a finding. Consequently, a custom mapping procedure was required for each SAST tool. Specifically, for each scan result, a list of findings was collected, along with a binary value representing whether a runtime error had occurred for the SAST tool. Each finding referenced the flagged source code file and line, and a list of the CWE weaknesses reported. Only findings that reported

at least one CWE-1000-applicable weakness were collected to focus on security-related weaknesses, as opposed to syntactic errors and code quality issues.

Two of the evaluated SAST tools, CodeQL and SonarQube, do not report CWE weaknesses in their findings, but instead report which pattern matching rule the source code matched. In the case of CodeQL, the CWE weaknesses associated with each rule were publicly documented [36] and were scraped from their website. For SonarQube, the associated CWE weaknesses for each rule were fetched via API calls to the `/api/rules/show` endpoint of the local SonarQube instance.

5.2.2 Mapping CWEs to CWE-1000 hierarchy

The hierarchical CWE View, CWE-1000, is used both to assess vulnerability detection efficacy and vulnerability classification quality of the SAST tools. In order to map each CWE weakness to its corresponding node in the CWE-1000 hierarchy, a local graph representation was built by referencing the descendant endpoint of the CWE REST API: `/api/v1/cwe/1000/descendants?view=1000`. The resulting graph representation was used in the evaluation to extract a list of ancestor nodes, hereafter referred to as a *Path*. The last ancestor of the Path is always a CWE Pillar, the top-most abstraction of CWE-1000. The ancestor CWE Pillar is used for the detection efficacy evaluation. If two CWEs share the same ancestor Pillar, it is considered a *Pillar match*. The Path is used to assess classification quality using the PF metric, introduced in section 2.4.

5.3 Evaluation metrics

With the collected SAST tool findings, a number of different metrics were produced to evaluate the tools. In particular, this section presents the metrics used to assess vulnerability detection efficacy, resource consumption and vulnerability classification quality, as well as details on how they were produced.

5.3.1 Vulnerability Detection

As discussed in section 2.4, traditional classification metrics such as the F1 score are less appropriate for evaluating on real-world vulnerabilities. Instead, following previous work [37, 40, 88], the number of detected vulnerabilities was used as a metric for vulnerability detection, expressed as a ratio.

Detection was measured in two levels of granularity: file-level detection and method-level detection. File-level detection considers a vulnerability detected if at least one vulnerable *file* is flagged as vulnerable. Similarly, method-level detection requires at least one vulnerable *method* to be flagged. In both cases, a file or method is considered vulnerable if it is modified in the vulnerability-fixing commit. Furthermore, detection was also measured with and without accounting for CWE Pillar classification. A detected vulnerability is considered to have the correct CWE Pillar if the SAST tool produced a vulnerability finding for the correct vulnerable file or method, and at least one of the reported CWE weaknesses shared the same ancestor CWE Pillar as the ground-truth CWE.

Following Li et al. [37], the number of vulnerabilities detected in the unpatched source code of each vulnerability-fixing commit was measured across four scenarios, reflecting all combinations of detection granularity and Pillar classification. The same terminology as Li et al. [37] is used for the detection scenarios to avoid confusion:

- **File-level Detection with Any CWE Pillar (S_{F-A}):** A vulnerability is detected if any vulnerable file is flagged, regardless of the reported Pillar CWE.
- **File-level Detection with Correct CWE Pillar (S_{F-C}):** A vulnerability is detected if any vulnerable file is flagged with the correct CWE Pillar.
- **Method-level Detection with Any CWE Pillar (S_{M-A}):** A vulnerability is detected if any vulnerable method is flagged, regardless of the reported CWE Pillar.
- **Method-level Detection with Correct CWE Pillar (S_{M-C}):** A vulnerability is detected if any vulnerable method is flagged with the correct CWE Pillar.

For each tool, scan level and detection scenario, the sets D_{vul} and D_{patch} were identified. These represent commits where a vulnerability was detected before and after applying the vulnerability-fixing commit, respectively. Following Li et al. [37], the approximate false positives were then determined as the commits where a vulnerability was detected both before and after applying the vulnerability-fixing commit:

$$FP = D_{\text{vul}} \cap D_{\text{patch}}$$

This approach to false positive approximation was chosen in favor of the ratio of marked functions due to it being equally applicable for file-level, commit-level and project-level scans. In contrast, the ratio of marked functions will be comparably high when scanning file-by-file and low for project-level scans, due to the varying number of files scanned.

Three metrics were finally presented:

- **Predicted Positives ($\boldsymbol{PP}_{\%}$)**: The ratio of predicted positives, more specifically the ratio of detected vulnerabilities before applying a vulnerability-fixing commit:
$$PP_{\%} = \frac{\#D_{\text{vul}}}{\#\text{Total Commits}}$$
- **True Positives ($\boldsymbol{TP}_{\%}$)**: The ratio of estimated true positives, defined as the number of detected vulnerabilities subtracted by the approximate false positives:
$$TP_{\%} = \frac{\#D_{\text{vul}} - \#FP}{\#\text{Total Commits}}$$

- **Precision (Pr)**: The ratio of estimated true positives among the detected vulnerabilities. A low precision implies more false positives among the detected vulnerabilities.

$$Pr = \frac{\#D_{\text{vul}} - \#FP}{\#D_{\text{vul}}}$$

Previous works [37, 88] only present the predicted positives, which may include false positives. Some tools report many findings for less critical code weaknesses, which gives them an unfair advantage in comparison. To address this limitation, the ratio of estimated true positives $TP_{\%}$ was presented in addition to the predicted positives.

Beyond detection metrics, the total number of findings produced by each SAST tool was also presented. Each finding corresponds to a flagged location in the source code, typically a line, associated with one or more CWE weaknesses. Consequently, a single file or method may contribute multiple findings if different lines are flagged.

5.3.2 Resource Consumption

Time was the sole metric employed for resource consumption evaluation. Time consumption was measured simultaneous to the evaluation of vulnerability detection efficacy, by measuring the completion time for each SAST tool scan. The time consumption was then evaluated in respect to the lines of code scanned, which in the case of file-level and commit-level scans was extracted from the `code_before` and `code_after` columns of the `file_change` table of CVEfixes. Since CVEfixes provides neither the total number of lines of code nor the full project source code, separate data collection was performed for each repository in the filtered dataset to determine project size in terms of lines of code and byte size.

5.3.3 Vulnerability Classification

Vulnerability classification quality was assessed as part of the SAST tool evaluation, using the PF score (See section 2.4). In short, the PF score quantifies how closely the predicted CWE classification aligns with the ground-truth CWE within the CWE-1000 hierarchy by measuring the fraction of shared ancestor nodes.

The original PF score (See equation 6.6) is defined as the average for each test case of the fraction of the predicted CWE Path that is also in the ground-truth CWE Path. This definition assumes there is only one predicted CWE Path per test case. Since the selected SAST tools can produce multiple CWE weaknesses for a single finding and multiple findings per vulnerability, the PF score was modified to accommodate for this. In particular, for each detected vulnerability, a CWE Path $\hat{Y}_{ji}$ of a CWE weakness reported among the modified files is chosen such that it maximizes the path fraction:

$$PF_{max} = \frac{1}{N} \sum_{j=1}^{N} \max_{i} \frac{\#(\hat{Y}_{ji} \cap Y_j)}{\#Y_j} \tag{5.1}$$

Here, N is the number of detected vulnerabilities and Y_j is the ground-truth CWE Path.

The modified PF score was measured for file-level (S_{F-A}) and method-level (S_{M-A}) detections, across all tools and scanning levels. It complements the binary Pillar match criteria used in S_{F-C} and S_{M-C}. Whereas the Pillar match criteria is a simple binary classification, the PF score provides a graded measure of the classification quality. A PF score of 1 indicates perfect classification and 0 means the predicted CWE weaknesses did not share a CWE Pillar with the ground-truth CWE.

5.4 Considerations

Due to initial estimates, performing scans on all of the code of the Java repositories was not plausible. Therefore the scans were limited to only the file changes in CVEfixes. This greatly reduces the scanning times but degrades the quality of the scans due to the limited scope. Scanning on only changes does not guarantee that the tools can see the source and the sink. The scanning times were shown to be greatly shorter than expected and scans for all of the code were ultimately chosen to be performed. Although the results of the limited scans does not show the tools' full capabilities, the results can still give some insight into what limitations the tools have when they are presented a limited scope.

Additional scans were originally also proposed. They would be based on the `method_change` table from the data set (see 4.1). The idea came from how tests had been performed by Atiiq et al. [7]. A file only containing a method is not valid Java syntax which caused some of the tools to not work. Injecting the method into an empty Java class does work, but variable called from outside of the method would not be present. If this information was to be included, it would not be very different from scanning the full file and is the reason why this level was dropped.

The file type filtering described under 4.3 was not performed for the project-level scans. While applying this filtering to project-level scans was considered, scanning entire projects without filtering provides a more accurate reflection of the time required for real-world scans.

Stated in 3.2, Semgrep supports cross-file analysis. To enable this, the tool requires the installation of the Semgrep pro engine and adding the `--pro` flag when scanning [66]. In addition to this, cross-file analysis must be activated online for the account used with the scanning [69]. Scans using this feature were tested, but due to unreasonably long scan times, it was chosen to be turned off. From estimates, the total runtime would be more than three weeks.

Chapter 6
Results

After more than 15 full days of CPU time, excluding time for cloning repositories and tool setup, all scans were complete. The resulting metrics are presented in this section, including the tools' vulnerability detection efficacy, time consumption and CWE-1000 classification accuracy.

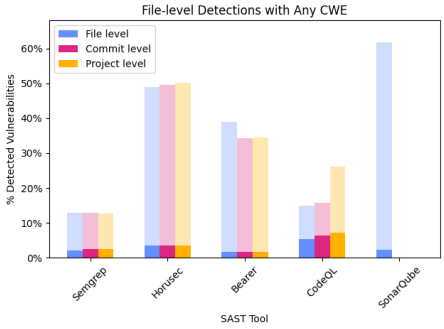
6.1 Vulnerability Detection Efficacy

Figure 6.1 shows the vulnerability detection efficacy results. Each subfigure displays the results for a separate detection scenario. The two figures on the top (6.1a, 6.1b) show the number of vulnerabilities flagged for file-level detection. The first one (6.1a) shows how many of the vulnerable files that were flagged with any CWE classification, and the second one how many of the vulnerable files had pillar matches. The figures at the bottom (6.1c, 6.1d) display the same statistics, but for method-level detection. For all of the graphs, approximate false positives are included as the dimmed part of the bars. Following Li et al. [37], vulnerabilities found both before and after a vulnerability-fixing commit are considered an approximation of false positives. The same data is represented in table 6.1.

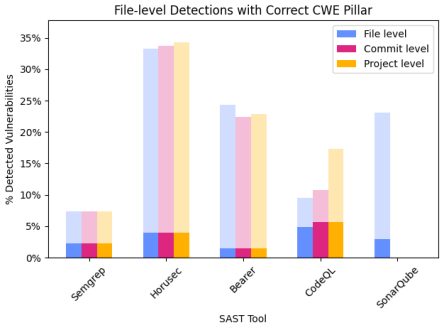
From the data in figure 6.1 and table 6.1, it can be seen that SonarQube flags the most vulnerabilities for *detections with any CWE*, but produces significantly fewer pillar matches in comparison to the other SAST tools. Apart from SonarQube, the relative standings of the tools in terms of predicted positives ($PP_{\%}$) are somewhat consistent across the different detection cases, with Horusec flagging the most, followed by Bearer, CodeQL and finally Semgrep. Subtracting the approximate false positives, as an approximate measure of true positives, it can be seen that CodeQL performs the best overall, followed by Horusec. Bearer performed the worst, with the least approximate true positives.

Between the different scanning levels, most of the tools were either stagnant or improving, both for predicted positives ($PP_{\%}$) and true positives ($TP_{\%}$) (see fig. 6.1). Bearer notably flags more vulnerabilities for file-level scans in all detection scenarios. Similarly, CodeQL flagged more vulnerabilities with each increase in scan size, most significantly for project-level scans.

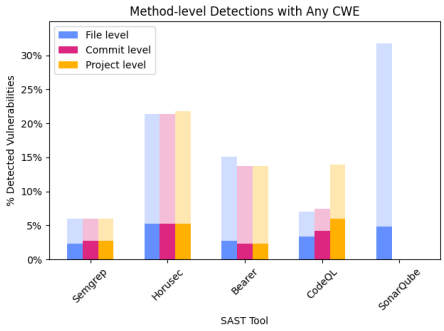
It can be observed in table 6.1 that the precision is greater for method-level detection than for file-level detection. In comparison with other SAST tools, CodeQL maintains the lowest false positive rate for the majority of detection cases. In



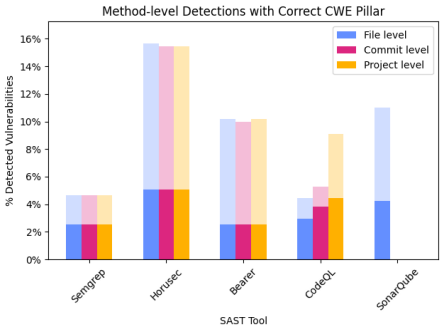
(a) S_{F-A}: The ratio of vulnerabilities where at least one vulnerable file was flagged, regardless of CWE reported.



(b) S_{F-C}: The ratio of vulnerabilities where at least one vulnerable file has a pillar match



(c) S_{M-A}: The ratio of vulnerabilities where at least one vulnerable method was flagged, regardless of CWE reported.



(d) S_{M-C}: The ratio of vulnerabilities where at least one vulnerable method has a pillar match

Figure 6.1: The vulnerability detection efficacy for each detection scenario. The bars represent the number of detected vulnerabilities $PP_{\%}$, whereas the bottom opaque parts represent the true positives $TP_{\%}$.

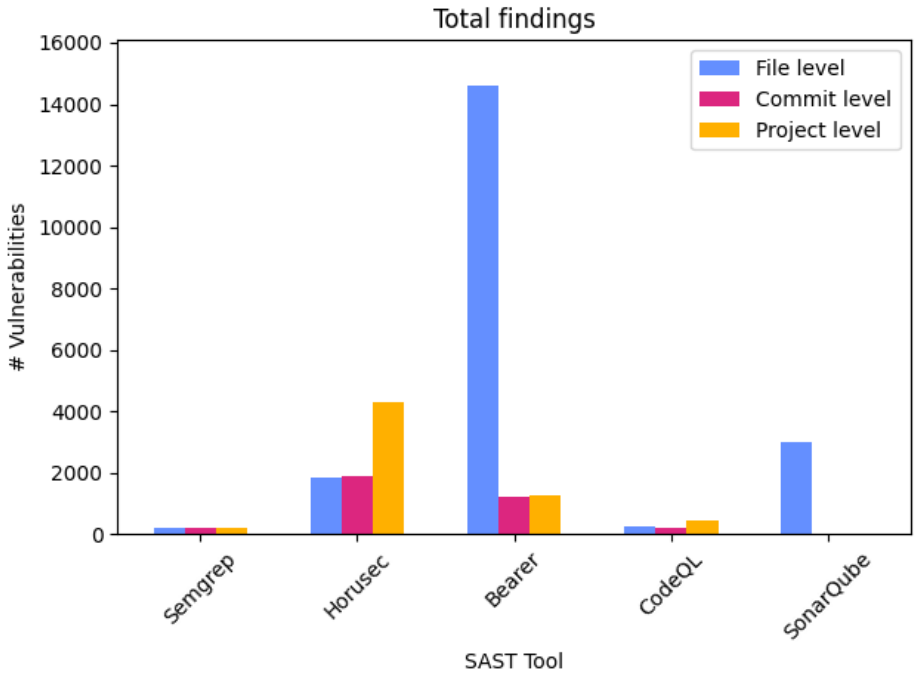


Figure 6.2: The number of total findings for each tool at different scan levels.

contrast, Bearer has the lowest precision for almost all scan levels and detection scenarios. While Horusec for the most part has the highest predicted positives ($PP_{\%}$), it has the second worst precision.

Looking at figure 6.2, it can be seen that Bearer produces a very large number of findings for file-level scans in comparison to the other tools. Horusec produced roughly double the amount of findings for project-level scans than for file- and commit-level scans. Through manual inspection, it seems that the large amount of Bearer findings for file-level scans could be partly attributed to duplicate findings, where the same rule was marked on multiple lines in a flagged file. Horusec and SonarQube also produced a larger number of findings than Semgrep and CodeQL, by a factor of approximately 10.

	Scan level	Metric	Bearer	CodeQL	Horusec	Semgrep	SonarQube
S_{F-A}	File	$PP_{\%}$	39.0%	14.8%	48.9%	12.9%	61.9%
		$TP_{\%}$	1.7%	5.3%	3.4%	2.1%	2.3%
		Pr	4.3%	35.7%	6.9%	16.4%	3.8%
	Commit	$PP_{\%}$	34.3%	15.7%	49.6%	12.9%	-
		$TP_{\%}$	1.7%	6.4%	3.4%	2.5%	-
		Pr	4.9%	40.5%	6.8%	19.7%	-
	Project	$PP_{\%}$	34.5%	26.1%	50.2%	12.7%	-
		$TP_{\%}$	1.7%	7.2%	3.4%	2.5%	-
		Pr	4.9%	27.6%	6.8%	20.0%	-
S_{F-C}	File	$PP_{\%}$	24.4%	9.5%	33.3%	7.4%	23.1%
		$TP_{\%}$	1.5%	4.9%	4.0%	2.3%	3.0%
		Pr	6.1%	51.1%	12.1%	31.4%	12.8%
	Commit	$PP_{\%}$	22.5%	10.8%	33.7%	7.4%	-
		$TP_{\%}$	1.5%	5.7%	4.0%	2.3%	-
		Pr	6.6%	52.9%	11.9%	31.4%	-
	Project	$PP_{\%}$	22.9%	17.4%	34.3%	7.4%	-
		$TP_{\%}$	1.5%	5.7%	4.0%	2.3%	-
		Pr	6.5%	32.9%	11.7%	31.4%	-
S_{M-A}	File	$PP_{\%}$	15.0%	7.0%	21.4%	5.9%	31.8%
		$TP_{\%}$	2.8%	3.4%	5.3%	2.3%	4.9%
		Pr	18.3%	48.5%	24.8%	39.3%	15.3%
	Commit	$PP_{\%}$	13.8%	7.4%	21.4%	5.9%	-
		$TP_{\%}$	2.3%	4.2%	5.3%	2.8%	-
		Pr	16.9%	57.1%	24.8%	46.4%	-
	Project	$PP_{\%}$	13.8%	14.0%	21.8%	5.9%	-
		$TP_{\%}$	2.3%	5.9%	5.3%	2.8%	-
		Pr	16.9%	42.4%	24.3%	46.4%	-
S_{M-C}	File	$PP_{\%}$	10.2%	4.4%	15.7%	4.7%	11.0%
		$TP_{\%}$	2.5%	3.0%	5.1%	2.5%	4.2%
		Pr	25.0%	66.7%	32.4%	54.5%	38.5%
	Commit	$PP_{\%}$	10.0%	5.3%	15.5%	4.7%	-
		$TP_{\%}$	2.5%	3.8%	5.1%	2.5%	-
		Pr	25.5%	72.0%	32.9%	54.5%	-
	Project	$PP_{\%}$	10.2%	9.1%	15.5%	4.7%	-
		$TP_{\%}$	2.5%	4.4%	5.1%	2.5%	-
		Pr	25.0%	48.8%	32.9%	54.5%	-

Table 6.1: Predicted positives, true positives and precision for the four detection scenarios. The values are presented for each detection scenario and tool.

6.2 Time Consumption

Figure 6.3 visualizes the time consumption of each tool for file-level scans. The running time of each tool appears to increase linearly with time. An exception to this is CodeQL with inconsistent results. Less data is available for Semgrep and CodeQL for the largest files. This is due to the way the scanning was set up, where file-level scans only initiated if vulnerabilities were found for the commit-level scans.

Figure 6.4 similarly visualizes the time consumption of each tool for commit-level scans. Once more, most of the tools seem to increase linearly. CodeQL is again an exception with sporadic results. Horusec, Bearer and Semgrep relatively retain the same rankings as for file-level scans. Note that SonarQube is not present due to its lack of commit-level scans.

Excluding CodeQL from the comparison, Horusec appears to run the fastest and Semgrep the slowest.

Figure 6.5 finally visualizes the time consumption for project-level scans. There are some significant outliers in terms of time, especially for CodeQL and Horusec. The largest ones for CodeQL were rescanned manually and verified to take a long time. Apart from these, the scanning time of CodeQL appears to act more linearly than for file- and commit-level scans. For the larger scans, the relative standings are different. Here, Bearer appears to be the fastest, followed by Semgrep, Horusec and CodeQL. In general, most of the tools seem to act less linearly than for file- and commit-level scans.

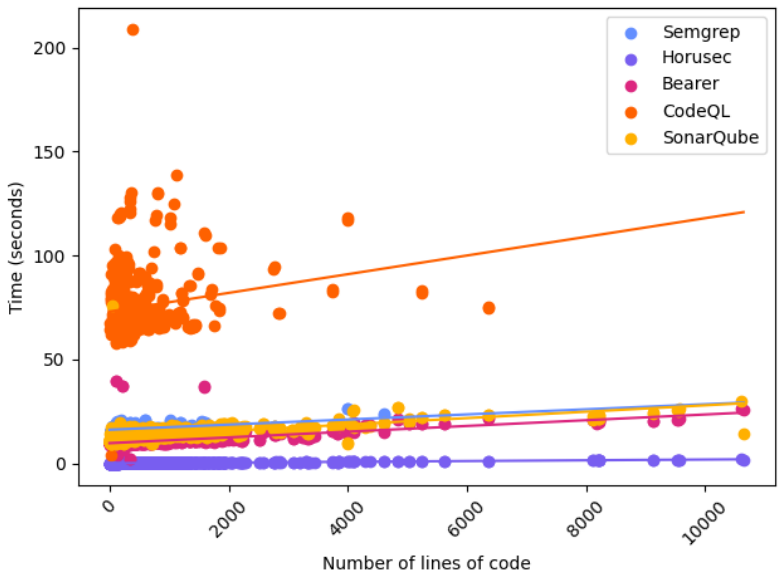


Figure 6.3: The file-level scan times at varying numbers of lines of Java code. Least square regression lines are included.

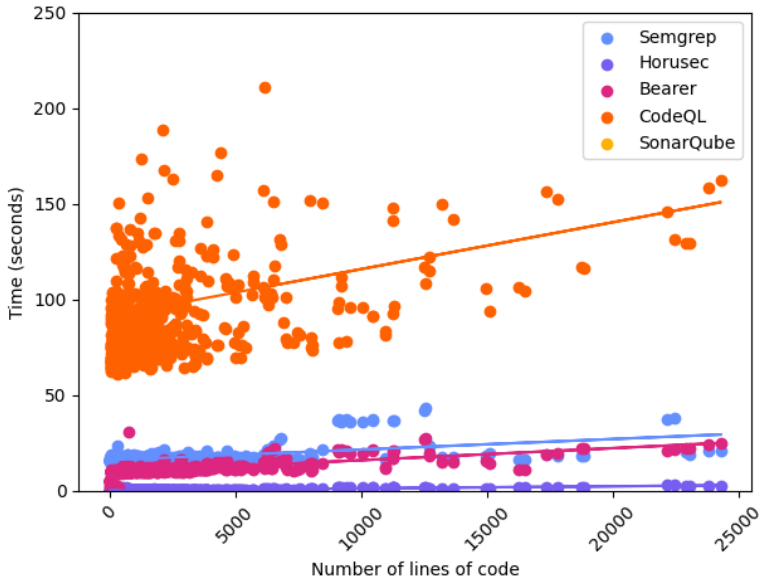


Figure 6.4: The commit-level scan times at varying total number of lines of Java code. Least square regression lines are included.

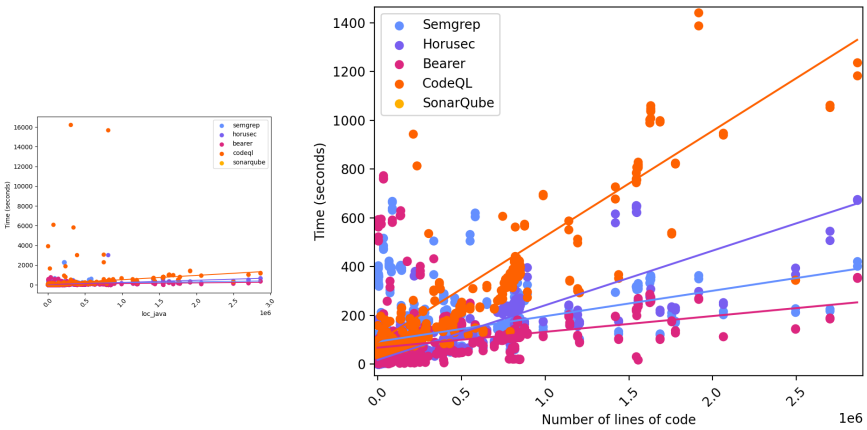


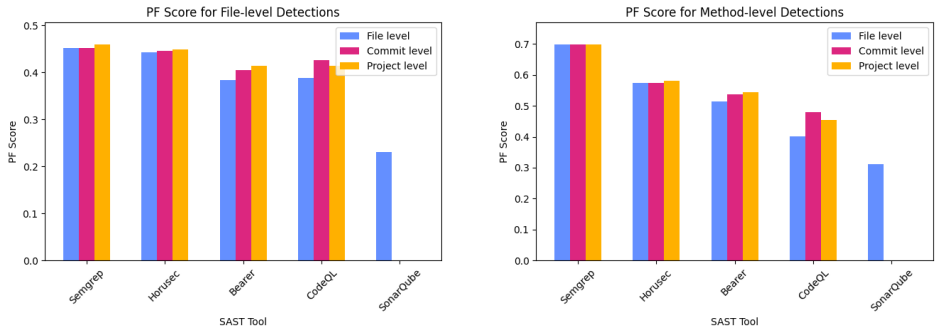
Figure 6.5: The project-level scan times for code repositories of varying total numbers of lines of Java code. The left figure shows the unzoomed view, including significant outliers, whereas the right shows a zoomed view focusing on the data surrounding the least square regression lines.

6.3 Vulnerability Classification

Figure 6.6a and 6.6b compares the PF scores for file-level and method-level detections with any CWE Pillar (S_{F-A} and S_{M-A}). Since SonarQube could only be scanned file-by-file, it has no data for the commit and project levels.

There is a clear difference in PF scores between file-level and method-level detections, with file-level detections achieving lower scores over all. PF scores were generally stable or improved across with an increase in scanning size, with the exception of CodeQL, which showed a decline for project-level scans.

Semgrep classified vulnerabilities best for both file-level and method-level detection, with average PF scores of 0.46 and 0.70 for file-level and method-level detection, respectively. Horusec followed close in second place. The PF scores for file-level detections are comparable for most tools, with the exception of SonarQube, which performs significantly worse in both detection scenarios. For method-level detections, the tools ranked clearly by performance, with PF scores in descending order: Semgrep, Horusec, Bearer, CodeQL and SonarQube.



(a) S_{F-A}: Vulnerabilities where at least one vulnerable *file* was flagged.

(b) S_{M-A}: Vulnerabilities where at least one vulnerable *method* was flagged.

Figure 6.6: PF scores for file-level and method-level detections with any CWE Pillar.

Chapter 7
Discussion

This chapter interprets the results presented in the evaluation, focusing on the strengths and limitations of the selected SAST tools. The first sections correspond directly to those in the evaluation chapter, covering vulnerability detection efficacy, time consumption and classification quality. The next section combines the findings and a conclusion is made regarding a recommended tool. The latter part of the chapter discusses broader theoretical considerations regarding the limitations of SAST tools and concludes with an overview of potential threats to the validity of the study.

7.1 Vulnerability Detection Efficacy

From the results, it is clear that Horusec and CodeQL are the main contenders in terms of vulnerability detection efficacy by their approximate true positive metrics. Li et al. [37] similarly found Horusec to be a top-performer, despite being a syntax-based tool with rules implemented by simple regular expression matching [37].

In contrast to Horusec, CodeQL goes beyond regular expressions, utilizing control-flow analysis and data-flow analysis, most notably taint analysis [37] to find vulnerabilities. It is unknown to what extent CodeQL is able to make use of taint analysis without building a project. Assuming the detection efficacy increases given access to the compiled code, CodeQL could surpass Horusec in performance in such case.

SonarQube noticeably flags significantly fewer vulnerabilities for the scenarios accounting for Pillar classification. Upon manual inspection, it was discovered that a significant portion of SonarQube's findings were categorized as *code smells*. These are not bugs, but rather warning signs of deeper issues that may increase the risk of bugs [75]. These code smells were not explicitly excluded, instead all findings with an associated CWE weaknesses were included in the evaluation. This approach was chosen for consistency across tools, but it turns out that SonarQube had a large portion of not directly security-related issues, which could affect results. Further inspection showed that these code smell findings also often map to a distinct CWE Pillar, namely *CWE-710: Improper Adherence to Coding Standards* [46] , which includes weaknesses such as *CWE-546: Suspicious Comment* [44] and *CWE-1041: Use of Redundant Code* [43]. The drop in detections for scenarios S_{F-C} and S_{M-C} could be explained by the fact that CWE-710 constitutes a very

small portion of the data set (See figure 4.2).

7.1.1 Precision

A large variation can be observed between the tools' precision. One explanation to this could be due to varying philosophies in vulnerability scanning. There is essentially a trade-off of the number of detected vulnerabilities and the number of false positives presented. Tools that aim for higher number of detections may flag a broader set of potential vulnerabilities to ensure that fewer true positives are missed, but at the cost of reporting more false positives. Tools optimized for precision instead tend to be more conservative in detections, reducing the burden of false positives, but potentially overlooking some genuine issues. This trade-off reflects differing priorities: security teams may prefer a higher number of detections to maximize coverage, while developers often value precision to minimize time spent reviewing findings. This is also reflected by the findings of Bennett et al. [9]: While reviewing false positives is a burden to developers, teams typically configure tools to reduce false negatives instead of false positives.

7.1.2 Performance Across Scanning Levels

Most SAST tools showed relatively consistent detection results across scanning levels. CodeQL's increase in efficacy for each increase in scan level could be explained by its novel technique in creating a database of the relational data of the entire code base, allowing for cross-file scans.

Both Semgrep and SonarQube also support cross-file scans. Since SonarQube was only performed on file-level, which means it only scanned files individually, cross-file scans are not possible. For Semgrep, as mentioned under 5.4, this feature was disabled due to long scan times. Each file is therefore scanned individually, which could explain why there is no increase in vulnerability detections between different scan levels.

One benefit with SAST tools not performing cross-file analysis is that the tools could be configured to scan only modified files, potentially dramatically decreasing the run time for use in development, whereas cross-file analysis needs to scan the entire code base to be effective.

Among Bearer's three scan levels, it had the highest number of detected vulnerabilities on file level, for all detection scenarios (see fig. 6.1). This is most likely related to Bearer's spike in findings for file level in figure 6.2. The reason to this behavior is still unknown as there is no documentation as to why this would happen.

7.1.3 Comparison with Previous Work

The project-level scan results for scenarios S_{F-C} and S_{M-C} were compared with the results of Li et al. [37] (See table 2.1). CodeQL demonstrates comparable performance across both evaluations. However, the remaining tools show notable discrepancies. In particular, Li et al. [37] report significantly higher precision for Horusec, reaching 71% in S_{M-C}, while maintaining a high number of detections. This thesis primarily uses default configurations for the evaluated tools,

whereas their evaluation explicitly enabled Horusec's integrated OWASP Dependency Check, which contributed an additional 49 detections. This configuration difference likely explains the performance gap. SonarQube, in contrast, reports fewer detections in their evaluation. Possibly, SonarQube was explicitly configured to exclude code smells, which could explain the difference in results, given that the default configuration used in this thesis includes them. Semgrep further shows more detections, possibly due to the rule set used, which could differ from the CI rule set used in this thesis. Additionally, differences in the dataset and filtering methods likely contribute to the variation in performance.

7.2 Time Consumption

The tools generally exhibited linear execution times across different scan levels, with the notable exception of CodeQL. The execution times were compared assuming linearity, based on claims by both CodeQL [26] and Semgrep [71] that their tools scale proportionally with the number of lines of code. However, as many of CodeQL's results significantly deviate from the regression line, it is reasonable to conclude that its behavior is not strictly linear. Manual inspection of the results (see fig. 6.3, 6.4, 6.5) indicates that the project-level scans (6.5) align most closely with the regression line. This suggests that CodeQL may tend toward linear execution times for larger projects than those included in this study.

Semgrep seems to be in a similar situation. File- and method-level scans align well with the regression line, but project-level scans resemble CodeQL's behavior. The deviating results at lower line counts might be the result from a higher proportion of non-Java files. Since project-level scans also scan the non-Java files present, perhaps a lot of projects at the lower end contain few Java files and a lot of other files.

Another tool that seems to be affected by scanning non-Java files is Horusec. In file- and commit-level scans, Horusec was clearly the fastest, with a noticeably flatter regression slope than the other tools. This changes quite a lot for the project-level scans where its running time is significantly higher than Semgrep and Bearer.

Since CodeQL is configured to only scan Java, non-Java files are ignored even given the additional context that scanning at project-level provides. Enabling support for more languages would likely further increase increase scan time.

A comparison with the execution times reported by Li et al. [37] reveals notable differences. As Li et al. [37] present no regression lines, it is difficult to conduct a direct comparison. Since different hardware was used, the times are different, but the relative scan times can still be compared. None of the tools match perfectly, but are not too far off, except Semgrep. Its execution time for Li et al. [37] is, comparatively speaking, significantly greater than either of the scan levels of this thesis. Whether Li et al. [37] are using the premium version with free access or the free version of Semgrep is unclear. If they use the premium version, they have access to cross-file analysis which significantly increases the scanning times. If they use the free version, there is a great chance that they are running Semgrep with the default configurations. This runs a project in a Docker container, adding to

the execution time. The use of any of these two features could explain why their execution time was greater. This highlights how tool configuration significantly affects execution time.

Running the tools with different configurations would not only change the execution times, but also their performance. The execution times alone are not sufficient for determining the best tool. For the tools that are based on rule sets, the execution times are closely tied to the size and complexity of the selected rule set. The `ci` rule set was chosen for Semgrep, which contains fewer rules than the `default` set. While the `default` rule set could increase execution time, it could also improve vulnerability detection, disregarding any possible impact from enabling cross-file analysis.

7.3 Vulnerability Classification

As observed, the PF scores for method-level detections are significantly higher than for file-level detections. This can intuitively be explained by the fact that method-level detections are less likely to include irrelevant findings and thus, the predicted CWE weakness align better with the ground-truth. By the same argument, the PF scores for method-level detections should be more representative in assessing CWE classification quality.

The low PF scores for SonarQube could be explained by code smells being included in its vulnerability findings, as discussed in section 7.1. As such, its PF score might not be fully representative of its classification quality. The same is not true for the remaining tools to the same extent.

Interestingly, while CodeQL performs well in detection scenarios accounting for Pillar classification (S_{F-C} and S_{M-C}), its PF score for method-level detections is worse than Bearer's and barely achieves a higher score than for file-level detections. This could indicate that CodeQL predicts less fine-grained CWE weaknesses in its findings, while still matching the correct Pillar CWE.

The stagnant or increase in PF score for each increase in scanning size aligns well with expectations. CodeQL achieving lower PF scores for project-level scans could be explained by the comparatively high number of detections for project-level scans, that are mostly attributed to false positives.

7.4 Recommended Tool

The evaluated tools showed varying performance. Determining what tool is the best depends very much on what metric is of importance. Despite this, Bearer can quite easily be ruled out as it has the worst overall performance among all of the tools. From table 6.1, it has the lowest rate of true positives and precision for almost all detection scenarios. SonarQube was the lowest performer in precision in cases where Bearer was not. If SonarQube would have been the worst performer if it had data for the commit- and project levels, is difficult to say without further investigation. Since SonarQube was configured to analyze source code as opposed to bytecode, it could not perform cross file analysis; performances would perhaps

be better for higher scan levels. With only the currently available data, its low precision, average scan time and low PF score make it difficult to recommend.

As cross-file analysis was also not utilized for Semgrep, its final results might not reflect its true potential. It had the second best precision results, even beating CodeQL in two cases. It however had the worst predicted positive rate which could be seen as insignificant if the precision rate is high. The high precision is likely a result from the choice of rule set, and choosing another rule set could yield a different precision. Its PF score is comparable to the other tools on file-level detection and has a significant lead for method-level detection. Although it generally had a short execution time, this would not be the case if cross-file analysis was enabled. Even if the performance could be better than the other tools, its long execution times could be prohibitively slow. Just as with SonarQube, Semgrep can not be recommended with the configurations used in this study.

From table 6.1, CodeQL and Horusec have the best results. While Horusec accounted for the majority of the highest predicted positive rates, CodeQL predominantly demonstrated superior precision, achieving percentages more than twice those of Horusec. As developers dislike high rates of incorrectly flagged vulnerabilities [9], CodeQL's dominance in precision could be determined to be more important than Horusec's dominance of predicted positives. Conversely, CodeQL takes at least double the time to scan compared to Horusec and has a notably worse PF. Based on these factors, it is difficult to prefer one of the tools over the other. Therefore, both **CodeQL and Horusec are chosen as the recommended tools**. However, since a lot of CodeQL's features were turned off in order to comply with the selected configuration criteria, CodeQL has the potential to outperform Horusec for more permissive configurations.

7.5 SAST Limitations

From table 6.1, it can be seen that the greatest true positive rate for any scan level is merely 7.2%. This gives an insight into the current limitations of SAST tools. As new vulnerabilities are continuously discovered, the CWE list evolves to reflect an ever-shifting threat landscape. SAST tools are therefore engaged in a constant race to keep pace; without regular updates and improvements, they risk falling behind and becoming ineffective against emerging security threats.

Analogous to the famous halting problem, determining whether arbitrary code is vulnerable is, in the general case, undecidable without execution. Since SAST tools rely exclusively on static analysis, meaning they examine code without it being executed, it raises fundamental challenges in achieving comprehensive vulnerability detection. Moreover, because the theoretical upper bound of SAST effectiveness is unknown, it remains difficult to assess how close current tools are to reaching their full potential.

To improve detection capabilities, SAST tools can incorporate more advanced techniques such as taint analysis, cross-file analysis or data-flow analysis. However, these approaches often lead to longer scan times, which may limit their practical feasibility in software development. This was seen with Semgrep, where its cross-file analysis feature made the execution time unbearably long. An alternative

direction is the integration of AI-based techniques. They have seen rapid developments and growing adoption over the past year and may offer a path toward more flexible and context-sensitive vulnerability detection.

7.6 Threats to Validity

One threat to validity lies in the definition of what constitutes a detected vulnerability. A SAST tool flagging a modified line from a vulnerability-fixing commit does not necessarily indicate correct identification; it could flag for an unrelated issue. To address this threat, Li et al. [37]'s approach to vulnerability detection was followed. It divides vulnerability detection in four well defined detection scenarios, incorporating CWE Pillar classification as a validating measure. However, to ensure validity, manual review of results would be required by experts in the field.

Another threat is the existence of undetected vulnerabilities in the data set used. To address this, only the detection efficacy of known vulnerabilities was evaluated.

A metric $TP_{\%}$ was introduced that approximates true positives by discarding vulnerabilities that are flagged both before and after a vulnerability-fixing commit. The validity of this metric lies in the assumption that no vulnerability detected both before and after the commit correctly refers to the fixed vulnerability.

An important threat to consider is the effect of varying tool configurations on detection efficacy. To minimize this risk, efforts were made to use the default configurations for all evaluated tools. However, as discussed in the previous section, certain features were deliberately disabled, most notably, analysis of compiled code and cross-file analysis specifically for Semgrep. These limitations were factored into the result analysis and tool recommendations, under the assumption that enabling such features would likely enhance detection efficacy. Varying tool configurations also have a significant impact on execution time, such as cross-file analysis being disabled for Semgrep. This was taken into account when interpreting the results, with the assumption that enabling more advanced features, such as cross-file analysis, generally leads to longer execution times.

Chapter 8
Conclusion

As there are many tools available on the market, it is difficult to know which one is the best. In this thesis, five different Java SAST tools were compared in terms of efficacy and execution time. Limitations were imposed on the tools to make them more comparable, but at the cost of not testing their full capabilities. Therefore major features such as cross-file analysis were not used which could drastically change the outcome of the study. Nonetheless, CodeQL and Horusec were determined to be the best tools, with a greater weight on efficacy over execution time. Despite both being chosen as the best, CodeQL might be the real winner, as it has unutilized features that Horusec does not support. To determine which tool is the best while utilizing the tools' full capabilities would require further research.

8.1 Future work

During the progress of the study, ideas of implementations to the work were proposed, but were never made due to time constraints. The following things were considered:

- **Complete feature utilization** - Allowing tools to use all of their features, displaying their full capabilities. This includes building projects, allowing SonarQube to run on all scan levels.
- **Free vs. Premium** - Testing free and premium setups for one tool to determine if it is worth paying for the premium features.
- **Pillar confusion matrix** - Generating a confusion matrix for pillar matches to determine which pillars are more often mixed up. This could be useful for SAST tool developers.
- **Evaluating more metrics** - Looking at other factors such as memory usage, ease of use or power consumption.
- **Historic performance** - Measure how SAST tools have improved over time. This could help in determining if SAST is still worth using or if other technologies should replace it.

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006. ISBN: 978-0321486813.

[2] Bushra Aloraini. "Towards better static analysis security testing methodologies". In: (2020). URL: http://hdl.handle.net/10012/16359.

[3] M. Alqaradaghi et al. "Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in Java code". In: *IEEE Access* 12 (2024), pp. 55824–55842. DOI: 10.1109/ACCESS.2024.3389955. URL: https://doi.org/10.1109/ACCESS.2024.3389955.

[4] Richard Amankwah et al. "Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites". In: *Software: Practice and Experience* 53.5 (2023), pp. 1125–1143. URL: https://doi.org/10.1002/spe.3181.

[5] A. S. Ami et al. ""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, 2024, pp. 19–23. DOI: 10.1109/SP.2024.00000. URL: https://doi.org/10.48550/arXiv.2307.16325.

[6] Andrew W. Appel. "Dataflow Analysis". In: *Modern Compiler Implementation in ML*. Cambridge University Press, 1997, pp. 377–403.

[7] Syafiq Al Atiiq et al. "From Generalist to Specialist: Exploring CWE-Specific Vulnerability Detection." In: (2024). URL: https://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edsarx&AN=edsarx.2408.02329&site=eds-live&scope=site.

[8] Bearer. *How Bearer CLI works*. Accessed: 2025-04-24. 2025. URL: https://docs.bearer.com/explanations/workflow/.

[9] Gareth Bennett et al. “Do Developers Use Static Application Security Testing (SAST) Tools Straight Out of the Box? A large-scale Empirical Study”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2024, pp. 454–460.

[10] Gareth Bennett et al. “Semgrep*: Improving the limited performance of static application security testing (sast) tools”. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 614–623. DOI: 10.1145/3661167.3661262. URL: https://doi.org/10.1145/3661167.3661262.

[11] G. Bhandari et al. “CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software”. In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*. 2021, p. 10. DOI: 10.1145/3475960.3475985. URL: https://doi.org/10.1145/3475960.3475985.

[12] Wachiraphan Charoenwet et al. “An empirical study of static analysis tools for secure code review”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024, pp. 691–703.

[13] Checkmarx. *CxSAST - Source Code Scanning*. Accessed: 2024-12-18. n.d. URL: https://checkmarx.com/cxsast-source-code-scanning/.

[14] Checkmarx. *CxSAST Release Notes for Version 9.6.0*. Accessed: 2024-12-18. n.d. URL: https://docs.checkmarx.com/en/34965-182362-release-notes-for-9-6-0.html.

[15] CodeQL. *About CodeQL*. Accessed: 2025-04-24. 2025. URL: https://codeql.github.com/docs/codeql-overview/about-codeql/.

[16] CodeQL. *About data flow analysis*. Accessed: 2025-04-24. 2025. URL: https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/.

[17] MITRE Corporation. *About the Common Weakness Enumeration (CWE)*. Accessed: 2024-12-09. n.d. URL: https://cwe.mitre.org/about/index.html.

[18] OpenText Corporation. *Fortify Static Code Analyzer*. Accessed: 2024-12-18. n.d. URL: https://www.opentext.com/products/fortify-static-code-analyzer.

[19] Mohamad Yusof Darus et al. "Enhancing Web Application Penetration Testing with a Static Application Security Testing (SAST) Tool". In: *2023 IEEE 8th International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*. IEEE. 2023, pp. 1–6. DOI: 10.1109/ICRAIE59459.2023.10468317.

[20] S. Elder et al. "Do I really need all this work to find vulnerabilities? An empirical case study comparing vulnerability detection techniques on a Java application". In: *arXiv preprint arXiv:2208.01595* (2022). DOI: 10.48550/arXiv.2208.01595. URL: https://doi.org/10.48550/arXiv.2208.01595.

[21] Patrick Engebretson. "Chapter 1 - What is Penetration Testing?" In: *The Basics of Hacking and Penetration Testing (Second Edition)*. Ed. by Patrick Engebretson. Second Edition. Boston: Syngress, 2013, pp. 1–18. ISBN: 978-0-12-411644-3. DOI: https://doi.org/10.1016/B978-0-12-411644-3.00001-7. URL: https://www.sciencedirect.com/science/article/pii/B9780124116443000017.

[22] Matteo Esposito et al. "An extensive comparison of static application security testing tools". In: *arXiv preprint arXiv:2403.09219* (2024). URL: https://doi.org/10.48550/arXiv.2403.09219.

[23] Inc. Facebook. *Infer Static Analyzer*. Accessed: 2024-12-18. n.d. URL: https://github.com/facebook/infer.

[24] Micro Focus. *Fortify Static Code Analyzer and Tools Documentation*. Accessed: 2024-12-18. n.d. URL: https://www.microfocus.com/documentation/fortify-static-code-analyzer-and-tools/2440/.

[25] OWASP Foundation. *OWASP Benchmark Project*. Accessed: 2025-01-30. 2021. URL: https://owasp.org/www-project-benchmark.

[26] GitHub. *Analysis takes too long - GitHub Docs*. Accessed: 2025-01-21. 2025. URL: https://docs.github.com/en/code-security/code-scanning/troubleshooting-code-scanning/analysis-takes-too-long.

[27] GitHub. *CodeQL - Analyze Code for Vulnerabilities and Errors*. Accessed: 2024-12-09. n.d. URL: https://github.com/github/codeql.

[28] GitHub. *CodeQL Zero to Hero Part 1: The Fundamentals of Static Analysis for Vulnerability Research*. Accessed: 2025-01-09. 2023. URL: https://github.blog/developer-skills/github/codeql-zero-to-hero-part-1-the-fundamentals-of-static-analysis-for-vulnerability-research/#early-static-analysis-tools-lexical-pattern-matching.

[29] Github. *static-analysis*. Accessed: 2025-01-07. URL: https://github.com/topics/static-analysis.

[30] Katerina Goseva-Popstojanova et al. "On the capability of static code analysis to detect security vulnerabilities". In: *Information and Software Technology* 68 (2015), pp. 18–33.

[31] Ahmad Hazimeh et al. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proc. ACM Meas. Anal. Comput. Syst.* 4.3 (Dec. 2020). DOI: 10.1145/3428334. URL: https://doi.org/10.1145/3428334.

[32] Horusec. *Overview*. Accessed: 2025-04-23. 2021. URL: https://docs.horusec.io/docs/cli/analysis-tools/open-source-horusec-engine/overview/.

[33] Zup IT Innovation. *Horusec - Security Static Analysis for Developers*. Accessed: 2024-12-18. n.d. URL: https://github.com/ZupIT/horusec.

[34] A. Kaur et al. "A comparative study of static code analysis tools for vulnerability detection in C/C++ and JAVA source code". In: *Procedia Computer Science* 171 (2020), pp. 2023–2029. DOI: 10.1016/j.procs.2020.04.217. URL: https://doi.org/10.1016/j.procs.2020.04.217.

[35] Hakan KEKÜL et al. "Comparison and analysis of software vulnerability databases". In: *International Journal of Engineering and Manufacturing* 12.4 (2022), p. 1.

[36] GitHub Security Lab. *CodeQL full CWE Coverage*. Accessed: 2025-04-14. 2025. URL: https://codeql.github.com/codeql-query-help/full-cwe.

[37] Kaixuan Li et al. "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 921–933. ISBN: 9798400703270. DOI: 10.1145/3611643.3616262. URL: https://doi.org/10.1145/3611643.3616262.

[38] Xiaozhou Li et al. "The anatomy of a vulnerability database: A systematic mapping study". In: *Journal of Systems and Software* 201 (2023), p. 111679.

[39] Johannes Link. "Chapter 1 - Introduction". In: *Unit Testing in Java*. Ed. by Johannes Link. The Morgan Kaufmann Series in Software Engineering and Programming. San Francisco: Morgan Kaufmann, 2003, pp. 3–21. ISBN: 978-1-55860-868-9. DOI: https://doi.org/10.1016/B978-155860868-9/50003-1. URL: https://www.sciencedirect.com/science/article/pii/B9781558608689500031.

[40] Stephan Lipp et al. "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*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 544–555. ISBN: 9781450393799. DOI: 10.1145/3533767.3534380. URL: https://doi.org/10.1145/3533767.3534380.

[41] Achmad Fahrurrozi Maskur et al. "Static Code Analysis Tools with the Taint Analysis Method for Detecting Web Application Vulnerability". In: *2019 International Conference on Data and Software Engineering (ICoDSE)*. 2019, pp. 1–6. DOI: 10.1109/ICoDSE48700.2019.9092614.

[42] Dan Mateer. *The Cost Savings of Fixing Security Flaws in Development*. Accessed: 2025-05-07. 2025. URL: https://www.hackerone.com/blog/cost-savings-fixing-security-flaws.

[43] Mitre. *CWE-1041: Use of Redundant Code*. Accessed: 2025-05-05. URL: https://cwe.mitre.org/data/definitions/1041.html.

[44] Mitre. *CWE-546: Suspicious Comment*. Accessed: 2025-05-05. URL: https://cwe.mitre.org/data/definitions/546.html.

[45] Mitre. *CWE-697: Incorrect Comparison*. Accessed: 2025-05-05. URL: https://cwe.mitre.org/data/definitions/697.html.

[46] Mitre. *CWE-710: Improper Adherence to Coding Standards*. Accessed: 2025-05-05. URL: https://cwe.mitre.org/data/definitions/710.html.

[47] Leon Moonen et al. *CVEfixes Dataset: Automatically Collected Vulnerabilities and Their Fixes from Open-Source Software*. Version v1.0.8. July 2024. DOI: 10.5281/zenodo.4476563. URL: https://doi.org/10.5281/zenodo.4476563.

[48] Anh Nguyen-Duc et al. "On the adoption of static analysis for software security assessment–A case study of an open-source e-government project". In: *Computers & Security* 111 (2021), p. 102470. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102470.

URL: https://www.sciencedirect.com/science/article/pii/S0167404821002947.

[49] Anh Nguyen-Duc et al. "On the combination of static analysis for software security assessment–a case study of an open-source e-government project". In: *arXiv preprint arXiv:2103.08010* (2021).

[50] *SARIF support for code scanning*. Accessed: 2025-05-25. URL: https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning.

[51] Tosin Daniel Oyetoyan et al. "Myths and facts about static application security testing tools: An action research at telenor digital". In: *Agile Processes in Software Engineering and Extreme Programming: 19th International Conference, XP 2018, Porto, Portugal, May 21–25, 2018, Proceedings 19*. Springer International Publishing. 2018, pp. 86–103.

[52] Shengyi Pan et al. "Fine-grained commit-level vulnerability type prediction by CWE tree structure". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 957–969.

[53] Yuanyuan Pan. "Interactive application security testing". In: *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*. IEEE. 2019, pp. 558–561.

[54] Serena Elisa Ponta et al. "A manually-curated dataset of fixes to vulnerabilities of open-source software". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 383–387.

[55] CVE Program. *About the CVE Program*. Accessed: 2025-03-24. 2025. URL: https://www.cve.org/About/Overview#AbouttheCVEProgram.

[56] CVE Program. *CVE Record Lifecycle*. Accessed: 2025-03-24. 2025. URL: https://www.cve.org/About/Process#CVERecordLifecycle.

[57] Bearer Project. *Bearer - Privacy and Security for Your Source Code*. Accessed: 2024-12-18. n.d. URL: https://github.com/Bearer/bearer.

[58] Find-Sec-Bugs Project. *Find Security Bugs*. Accessed: 2024-12-18. n.d. URL: https://find-sec-bugs.github.io/.

[59] PMD Project. *PMD - An extensible cross-language static code analyzer*. Accessed: 2024-12-18. n.d. URL: https://pmd.github.io/.

[60] Semgrep Project. *Semgrep - Lightweight Static Analysis for Many Languages*. Accessed: 2024-12-18. n.d. URL: https://github.com/semgrep/semgrep.

[61] Contrast Security. *Contrast Scan Release Notes and Archive*. Accessed: 2024-12-18. n.d. URL: https://docs.contrastsecurity.com/en/scan-release-notes-and-archive.html.

[62] Contrast Security. *Contrast Security Documentation*. Accessed: 2024-12-18. n.d. URL: https://docs.contrastsecurity.com/index.html?lang=en.

[63] Insider Security. *Insider - Static Application Security Testing (SAST)*. Accessed: 2024-12-18. n.d. URL: https://github.com/insidersec/insider.

[64] Semgrep. *Announcing SonarQube Advanced Security*. Accessed: 2025-04-23. 2025. URL: https://www.sonarsource.com/blog/announcing-sonarqube-advanced-security/.

[65] Semgrep. *ci*. Accessed: 2025-04-14. URL: https://semgrep.dev/p/ci.

[66] Semgrep. *Cross-file analysis examples*. Accessed: 2025-04-24. 2024. URL: https://semgrep.dev/docs/semgrep-code/semgrep-pro-engine-examples.

[67] Semgrep. *Cross-file analysis taint traces*. Accessed: 2025-04-24. 2025. URL: https://semgrep.dev/docs/semgrep-code/semgrep-pro-engine-data-flow.

[68] Semgrep. *default*. Accessed: 2025-04-14.

[69] Semgrep. *Perform cross-file analysis*. Accessed: 2025-04-24. 2024. URL: https://semgrep.dev/docs/semgrep-code/semgrep-pro-engine-intro.

[70] Semgrep. *Perform cross-file analysis*. Accessed: 2025-04-23. 2025. URL: https://semgrep.dev/docs/semgrep-code/semgrep-pro-engine-intro.

[71] Semgrep. *Performance Principles for Rule Files - Semgrep*. Accessed: 2025-01-21. 2025. URL: https://semgrep.dev/docs/kb/rules/rule-file-perf-principles#:~:text=Generally%2C%20the%20time%20required%20to,setup%20work%20%2B%20time%20for%20matching..

[72] Semgrep. *Supported languages*. Accessed: 2025-04-24. 2025. URL: https://semgrep.dev/docs/supported-languages.

[73] Semgrep. *Taint analysis*. Accessed: 2025-04-24. 2025. URL: https://semgrep.dev/docs/writing-rules/data-flow/taint-mode.

[74] Semgrep. *What is 'taint analysis' and why do I care?* Accessed: 2025-04-24. 2025. URL: https://www.sonarsource.com/blog/what-is-taint-analysis/.

[75] Semgrep. *What is a code smell?* Accessed: 2025-05-25. n.d. URL: https://www.sonarsource.com/learn/code-smells/.

[76] Snyk. *Snyk CLI*. Accessed: 2024-12-18. n.d. URL: https://github.com/snyk/cli.

[77] SonarQube. *Cognitive Complexity, Because Testability != Understandability*. Accessed: 2025-04-25. 2016. URL: https://www.sonarsource.com/blog/cognitive-complexity-because-testability-understandability/.

[78] SonarQube. *The Power of Taint Analysis: Uncovering Critical Code Vulnerability in OpenAPI Generator*. Accessed: 2025-04-25. 2024. URL: https://www.sonarsource.com/blog/the-power-of-taint-analysis-uncovering-critical-code-vulnerability-in-openapi-generator/.

[79] SonarSource. *SonarQube*. Accessed: 2024-12-18. n.d. URL: https://github.com/SonarSource/sonarqube.

[80] National Institute of Standards et al. *SARD Test Suites - Suite 111*. Accessed: 2025-01-08. 2017. URL: https://samate.nist.gov/SARD/test-suites/111.

[81] CWE Content Team. *CVE → CWE "Root Cause Mapping" Guidance*. Accessed: 2025-03-27. 2024. URL: https://cwe.mitre.org/documents/cwe_usage/guidance.html.

[82] CWE Content Team. *CWE VIEW: Deprecated Entries*. Accessed: 2025-01-22. 2007. URL: https://cwe.mitre.org/data/definitions/604.html.

[83] CWE Content Team. *CWE VIEW: Research Concepts*. Accessed: 2025-02-04. 2008. URL: https://cwe.mitre.org/data/definitions/1000.html.

[84] CWE Content Team. *CWE VIEW: Weaknesses in OWASP Top Ten (2021)*. Accessed: 2025-03-25. 2021. URL: https://cwe.mitre.org/data/definitions/1344.html.

[85] CWE Content Team. *CWE VIEW: Weaknesses in the 2024 CWE Top 25 Most Dangerous Software Weaknesses*. Accessed: 2025-03-25. 2024. URL: https://cwe.mitre.org/data/definitions/1430.html.

[86] Ferdian Thung et al. “To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 50–59.

[87] Xin Zhou et al. *Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection*. 2024. arXiv: 2407.16235 [cs.SE]. URL: https://arxiv.org/abs/2407.16235.

[88] Xin Zhou et al. “Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead”. In: *ACM Trans. Softw. Eng. Methodol.* (Dec. 2024). Just Accepted. ISSN: 1049-331X. DOI: 10.1145/3708522. URL: https://doi.org/10.1145/3708522.