Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Analysis of tools for static security testing of applications

Analysis of SAST tools

Leonid Golovyrin

Supervisor: Ing. Josef Kokeš, Ph.D.
Field of study: Open Informatics
Subfield: Software
May 2023

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ing. Josef Kokeš, Ph.D., for his guidance, valuable advice, and I am particularly grateful for the opportunity he gave me to explore this challenging and rewarding topic.

Additionally, I would like to extend my appreciation to my friends and colleagues for sharing their knowledge and providing constructive feedback.

Finally, I am grateful for the unwavering support of my family, who have always been there for me.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 20, 2023

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 20. května 2023

Abstract

This thesis presents a comprehensive evaluation of general-purpose Static Application Security Testing (SAST) tools available to the general public and offering free versions.

The study focused on the integration of these tools into the Software Development Life Cycle (SDLC) by identifying and comparing top-performing tools such as CodeQL, Semgrep, and SonarQube.

An easily extensible rule analysis framework was employed to support the examination of their distinct features such as the metrics of their rules.

A practical SAST project was implemented using GitLab, demonstrating the integration of multiple SAST tools. This project was then leveraged to compare the default configurations of the SAST tools. The analysis clarified the precision of the tools and their tendency to generate false positives. Additionally, the usability of custom rules was assessed.

The study offers valuable insights and recommendations, helping to facilitate the selection and adoption of SAST tools.

Keywords: Static analysis, SAST, DevSecOps, CodeQL, Semgrep, SonarQube, Joern

Supervisor: Ing. Josef Kokeš, Ph.D.
Department of Information Security
FIT

Abstrakt

Tato bakalářská práce představuje analýzu nástrojů pro statické testování zabezpečení aplikací (SAST) obecného účelu, které jsou dostupné široké veřejnosti a nabízejí bezplatné verze.

Studie se zaměřuje na integraci těchto nástrojů do životního cyklu vývoje softwaru (SDLC) identifikací a porovnáním nejlepších nástrojů, jako jsou CodeQL, Semgrep a SonarQube.

Pro zkoumání jejich odlišných funkcí byl navržen a použit snadno rozšiřitelný framework analýzy metrik pravidel.

Za použití platformy GitLab byl implementován praktický SAST projekt, který demonstruje integraci vybraných SAST nástrojů. Tento projekt byl následně využit pro srovnání výchozích konfigurací SAST nástrojů. Analýza ukázala rozdíly v přesnosti detekce zranitelností a generování falešně pozitivních výsledků. Bylo také provedeno hodnocení použitelnosti funkcionality vlastních pravidel.

Studie nabízí cenné poznatky a doporučení, která usnadňují výběr a přijetí vhodných SAST nástrojů.

Klíčová slova: Statická analýza, SAST, DevSecOps, CodeQL, Semgrep, SonarQube, Joern

Překlad názvu: Analýza nástrojů pro statické testování bezpečnosti aplikací — Analýza SAST nástrojů

Contents

Figures

Tables

Chapter 1

Introduction

Around the world, digitalization is a driving force for economic growth [1]. Entire businesses, governments, and even critical infrastructures are becoming digitalized. The IT industry is growing rapidly and many aspects of modern society increasingly rely on digital technologies. To protect these vital systems from compromise and prevent the loss of sensitive data, it is crucial to implement security measures that minimize the attack surface and prevent the potential exploitation of vulnerabilities.

Cyberattacks are becoming more frequent and sophisticated [2], meaning security must keep pace. As software vulnerabilities are more likely to be discovered and exploited by attackers than hardware vulnerabilities [3], they should be the primary target for security hardening.

Securing software involves implementing the Secure Software Development Life Cycle (SSDLC) [4]. As part of the SSDLC "development" stage, static code analysis must be integrated [5]. Furthermore, static code scanning is recommended or required by many security standards and regulations [6]. Therefore, the need to analyze tools capable of performing static analysis arises.

1.1 Background

Securing software is a complex problem with many possible approaches, making it time-consuming to analyze and select the best options for each stage of the SSDLC. One potential strategy is to utilize existing knowledge and tools to address every aspect of software security, but this may not be simple. Some solutions are also dependent on the technologies already in use and require advanced integration into existing, and often highly specific, workflows. This is particularly true for automated static code analysis or more generic static application security testing (SAST), as effective use of SAST requires deep integration into the development process and tools [7].

1.2 Reasons

Identifying the most appropriate SAST tool is a complex task that requires in-depth knowledge of the tool's capabilities and the potential for integration into existing workflows. Certain tools are designed for specific development environments, making integration into different ones challenging and necessitating a longer adoption period to overcome obstacles. Even large companies like Google have faced challenges integrating SAST tools into their development workflows [8]. Some tools have limited abilities to detect security vulnerabilities in certain programming languages, so choosing them may result in overlooking common and damaging attack vectors [9].

Additionally, if SAST integration is not properly managed, the entire process may become a burden for the development teams, leading to less convenient security practices. Since lightweight security best practices play a crucial role in ensuring that compliance with security measures does not overburden the team [10], the intention to improve software security by using misconfigured or ill-suited tools could ultimately waste time and resources.

1.3 Objectives

The main objective of this thesis is to provide an overview of commonly used general-purpose SAST tools, compare their functionality and limitations. The second objective is to analyze the suitability of these tools for integration into a typical development life cycle, with a focus on the integration effort and future maintainability of the solution.

1.4 Delimitations

The scope of this thesis is limited to the analysis of the SAST tools that are available for free or under a permissive license. This also means that the tools that have some paid features will be analyzed only in the context of their free features.
The integration of the SAST tools into the development workflow is analyzed from the perspective of the GitLab CI/CD. Due to the variety of different programming languages available, Python is used for the examples in this thesis, while the analysis of the tools is done in a more general way.

Chapter 2

Application security theory

Before proceeding to the analysis and performing a comparison of the tools, we will provide a brief overview of some important concepts that will be used in the rest of the thesis.

2.1 The phenomenon of vulnerabilities

The vulnerabilities by themselves are only an unexpected behavior of the software; what makes them dangerous is the fact that they can be exploited by attackers to gain unauthorized access to the system or to steal sensitive data. Developers approach the problem of vulnerabilities in different ways, but generally, they are focused on correctly implementing the desired functionality, not on writing secure code [10]. The ability to write secure code is a skill that is not taught in most of the software engineering curricula [11], so it is not surprising that developers are often not aware of the security implications of their code.

2.2 Vulnerability scoring systems

Common Weakness Enumeration (CWE) helps to categorize the vulnerabilities and to provide a common vocabulary for describing them, making it easier to compare the results of different tools or to target some vulnerabilities as a whole class. However, it's crucial to remember that each vulnerability, despite being under the same CWE ID, may present unique characteristics and thus necessitate a distinct approach. It is not rational to distribute available resources uniformly across all vulnerabilities grouped under the same CWE ID, as their risk levels can considerably vary. For instance, there could be vulnerabilities within the same CWE class that are currently unexploitable and thus present a lower risk.

Vulnerabilities can be classified for two different reasons:

- prioritization of discovered vulnerabilities by their impact, or
- development of an overview of all common vulnerabilities and which ones are the most serious ones at the moment.

The Common Vulnerability Scoring System (CVSS) is as an industry-standard prioritization language that made it possible to effectively assess the severity and prioritize already discovered vulnerabilities [12].

Nowadays, the latest major revision CVSSv3 of the CVSS scoring mechanism is used, since it was designed as a successor of CVSSv2 to overcome previous shortcomings like implicit guidelines and excessively overwhelmed impact metrics [13].

CVSSv3 assigns a score to a specific vulnerability using multiple group groups of metrics:

- *Base metric group* represents the characteristics of a vulnerability that are constant over time and across environments, thus making it useful for many vulnerabilities rankings to be based exclusively on the base score, ignoring the other non-constant metrics.
  Base metrics are composed of two sets of metrics:
  - Exploitability metrics (Attack Vector, Attack Complexity, Privileges Required, User Interaction).
  - Impact metrics (Confidentiality, Integrity, Availability).
- *Temporal metric group* reflects changing factors outside of the user environment, such as the presence of a working proof of concept exploit or the existence of a patch.
- *Environmental metric group* represents the characteristics of vulnerability which are specific to the user environment, such as the presence of mitigation mechanisms that reduce the impact of the vulnerability or the relative importance of the affected component to the overall system.

In order to make the CVSS score more understandable for the less technical audience, the CVSS score can be also translated into a qualitative severity rating according to the following table:

Rating	CVSS Score
Critical	9.0 – 10.0
High	7.0 – 8.9
Medium	4.0 – 6.9
Low	0.1 – 3.9
None	0.0

Table 2.1: CVSS Qualitative Severity Rating Scale [13]

The Open Web Application Security Project (OWASP) is a non-profit organization that is focused on improving the security of software. It maintains a list of the most common vulnerability categories, which is called the OWASP Top 10 and is updated every two to four years [14]. To achieve the maximum real-world relevance of the list, many factors are taken into account while creating the OWASP Top 10 [14]:

- The number of CWEs mapped to a category.
- The Impact and Exploit sub-scores from CVSS scores assigned to CVEs related to a category.
- Occurrence statistics of the category in real applications, such as the total number of applications vulnerable to that CWE, the percentage of vulnerable applications among the tested ones, and the total number of CVEs.
- The percentage of applications tested by all organizations for a given CWE.

The latest version of the OWASP Top 10 list is from 2021, and it contains the categories shown in Table 2.2.

Category	Name
A01:2021	Broken Access Control
A02:2021	Cryptographic Failures
A03:2021	Injection
A04:2021	Insecure Design
A05:2021	Security Misconfiguration
A06:2021	Vulnerable and Outdated Components
A07:2021	Identification and Authentication Failures
A08:2021	Software and Data Integrity Failures
A09:2021	Security Logging and Monitoring Failures
A10:2021	Server-Side Request Forgery

Table 2.2: OWASP Top 10 2021 categories [14]

2.3 Different vulnerability types

The ability to identify whether a code is vulnerable to a specific type of attack cannot be accomplished exclusively by automatic tools since there are some vulnerabilities that may be hard to reliably detect automatically and manual verification is still required. Moreover, to achieve the full potential of SAST tools and use their advanced features, it is necessary to understand the different types of vulnerabilities and how they can occur in the code.

The vulnerability types described in the following sections will be used during the evaluation of the SAST tools; all of them can also be found in the OWASP Top 10 list.

2.3.1 Security Misconfiguration

Notable CWEs included: *Configuration (CWE-16)* and *Improper Restriction of XML External Entity Reference (CWE-611)*.

As software becomes highly configurable, it is important to ensure that the application is properly configured and does not contain any default credentials, unnecessary features or insecure configurations. There are several ways how this type of vulnerability can occur [15]:

- *Unhardened environment* – missing security hardening of application stack components or misconfigured permissions on cloud services.
- *Default credentials* – the application still accepts default credentials that are not changed or removed after installation.
- *Unnecessary functionality* – some features that should not be used in production are enabled, such as the debug mode, unnecessary ports or API endpoints.
- *Mismanaged updates* – the upgraded system does not properly reflect the latest security changes of the performed update.

2.3.2 Injection

Notable CWEs included: *Cross-site Scripting (CWE-79)*, *SQL Injection (CWE-89)* and *External Control of File Name or Path (CWE-73)*.

Injection vulnerabilities occur when the application passes untrusted data to an interpreter as a part of a command or query without sufficient prior validation, filtration, sanitization, or context-aware escaping [16]. Most common injections are SQL, NoSQL, ORM, OS command, LDAP, and Code injections [14].

2.3.3 Broken Access Control

Notable CWEs included: *Exposure of Sensitive Information to an Unauthorized Actor (CWE-200)*, *Insertion of Sensitive Information into Sent Data (CWE-319)* and *Cross-Site Request Forgery (CWE-352)*.

Broken access control vulnerabilities occur when the application fails to properly limit user actions according to their intended privileges. The application might be vulnerable to this type of attack if it has one of the following issues [17]:

- *Violating the principle of least privilege or deny by default* – the application grants more privileges to users than they need to perform their job.
- *Improperly configured access control* – the application does not properly verify the user's access rights, therefore allowing bypassing of the access control mechanism.
- *Insecure direct object references* – users are allowed to access objects directly by ID without any authorization checks.
- *CORS misconfiguration* – the application allows API requests from unintended origins.

2.3.4 Server Side Request Forgery

Notable CWEs included: *Server-Side Request Forgery (CWE-918)*.

Server Side Request Forgery (SSRF) vulnerabilities occur when the application allows to send attacker-controlled requests to internal services or other resources that are not intended to be externally accessible. The application might be vulnerable to this type of attack in the following cases [15]:

- *Violating the principle of deny by default* – set of accessible resources by the application is not restricted at the network level, or the application is not using an allowlist for the resources that are allowed to be accessed.
- *Mishandeling redirects* – the application allows resource validation bypass using request redirection.
- *URL consistency unawareness* – the application assumes that the URL always points to the same resource, ignoring the possibility of race conditions such as DNS rebinding.

2.4 Manual security testing

Manual security testing is a process that aims to identify potential security risks in software and implement measures to mitigate those risks. The "white-box" approach in software testing denotes methods where internal software components such as its source code or architecture are known and accessible. In contrast, "black-box" methods focus on software's functionality from an external view, with no insights into its internal logic or components. Several methods of manual security testing are used [18]:

- Code Review: During this process, the software's source code is examined to identify any potential security flaws. Code review is the most commonly used method in the modern software development industry. However, its main drawback is that it can be time-consuming and requires substantial expertise, making the code review process both costly and susceptible to errors [19].
- Penetration Testing: This method assesses the system for possible vulnerabilities that could be exploited by external threats. Ethical hackers or security experts usually perform penetration testing.
- Security Audit: This independent evaluation of a system's security posture is often conducted by a third party. A security audit can also assess a system's compliance with security standards and provide recommendations for achieving compliance.
- Threat Modeling: This process involves identifying, analyzing, and addressing potential threats to a system.

2.5 Automated security testing

Automated security testing (AST) is the process of automating the identification and exploitation of potential security vulnerabilities in software. AST can be classified into the following categories [20]:

- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Interactive Application Security Testing (IAST)
- Software Composition Analysis (SCA)

SAST, a white-box testing method, analyzes the application's source code, binaries, or compiled code to identify security vulnerabilities. Unlike linters – tools that flag syntax errors, bugs, and suspicious constructs in the source code, SAST uses knowledge of the application's internals to detect security issues. SAST tools are often integrated into Continuous Integration (CI) systems or as plugins in Integrated Development Environments (IDEs) for immediate feedback. By integrating SAST into the development workflow, vulnerabilities are prevented from reaching the production environment. This is a significant advantage over other types of security testing, as security vulnerabilities discovered later in the development cycle are more expensive to fix than those discovered early [21].

DAST is a black-box testing method in which the application is analyzed by running it in a live environment and interacting with it to identify security vulnerabilities. These security tests are performed by simulating attacks from the perspective of an external attacker, which limits DAST's ability to detect vulnerabilities that are not yet exploitable.

IAST is a hybrid testing method that combines the main features of SAST and DAST. A special agent records the application's state during testing in a live environment. The collected metrics and access to the source code allow the agent to provide more detailed information about the application's behavior and verify its findings.

SCA is a software testing technique that analyzes software libraries, components, and dependencies for known security vulnerabilities. This type of testing, also known as dependency scanning, is usually performed for the open-source parts of the application stack.

Chapter 3

SAST tools

Many open-source and proprietary SAST solutions have been developed to perform static code analysis for security vulnerabilities. Although the main feature of SAST is to detect security vulnerabilities in the source code, it is important to note that having the most advanced analysis engine is not the only important feature. In this chapter, we will highlight the important aspects of SAST tools that should be considered when choosing the most suitable tool for a particular project.

3.1 Overview of SAST workflow

From the user's perspective, the SAST functionality can be divided into three main parts:

- **Analysis configuration** – the ability to configure the analysis to the needs of the project.
- **Source code analysis** – the main part of the SAST functionality that scans the source code and detects potential security vulnerabilities.
- **Findings management** – the part of the SAST functionality that is responsible for presenting the results of the source code analysis to the user and tooling to mitigate the detected findings.

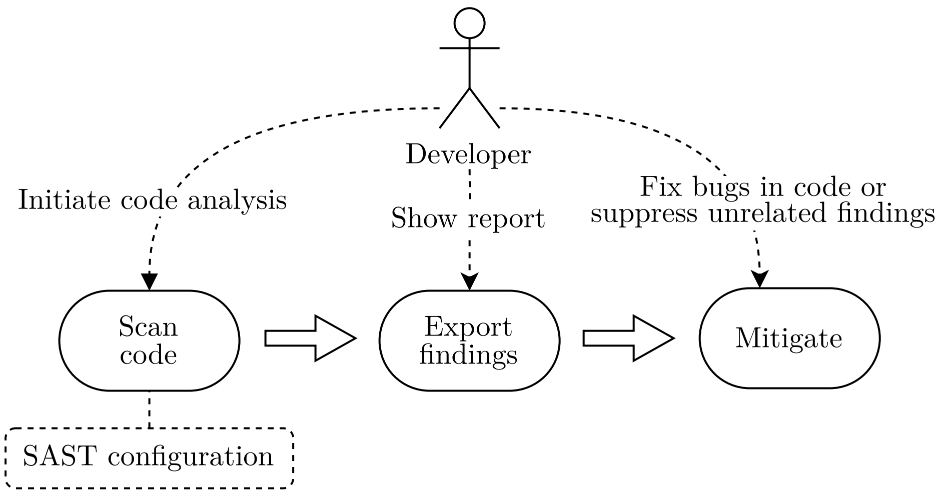


Figure 3.1: SAST workflow cycle.

3.1.1 Analysis configuration

Every project may differ in terms of the programming language, the libraries and the guidelines that are used during the development. Therefore, even the same line of code can be considered a security vulnerability in one project and not in another. With this in mind, it is important to configure the SAST tool to match the project's needs as much as possible.

That can be done by carefully selecting the rules that will be used during the scan. Most SAST tools provide a set of rules that are as generic as possible while still covering the most common vulnerabilities, so users can start using the tool right away and then gradually refine the rules to achieve the desired level of accuracy.

But even with the large selection of rules, there are still a lot of possible project-specific cases where the ability to create custom rules may be necessary. For example, if the project uses an internal library that is not publicly available, forcing the tool to generate false positives for the library's code that is not under the user's control.

3.1.2 Source code analysis

Although the idea behind the code analysis remains the same for all SAST tools, this part of the SAST functionality is generally implemented using various approaches. Therefore, depending on the chosen SAST tool, different behavior in terms of the tool's speed and accuracy may be observed. More on this topic can be found in the Section 3.3.2.

3.1.3 Findings management

After an analysis had been performed, the results are converted into a report that can be used during the mitigation phase.

All the findings generated can be divided into two main categories:
False positives (FP) – findings that are not actually security vulnerabilities, thus they should be mitigated on the tool's side, and *True positives (TP)* – findings that may represent a security vulnerability and should be mitigated by the project's developers.
While dealing with the FP findings, the functionality to easily suppress a rule generating these findings for specific blocks of code or files may be found useful so that the tool will not generate the same finding again and does not contribute to the most disturbing statistic in the SAST world – the number of false positives [22].
Another option, especially for large projects, is to integrate the SAST tool with the Vulnerability Management System (VMS) such as open-source *DefectDojo* [1] or commercial *ThreadFix* [2] and *Nucleus* [3]. This allows the tool to directly report the findings to the VMS and provides a centralized workflow for managing findings, including FP suppression.

3.2 Integration methods

The detection and mitigation phases of the SAST workflow are highly dependent on how the tool is integrated into the development process. Various integration points must be considered, including the scanning environment, tool configuration distribution and collaboration with the VMS.

SAST tools are commonly integrated into CI/CD pipelines as cloud-based scanning offers several benefits, most notably the great performance improvements, centralization of findings and rules management [23].

[1] https://owasp.org/www-project-defectdojo/
[2] https://www.coalfire.com/solutions/application-security/threadfix
[3] https://nucleussec.com/

This allows the scanning CI/CD job to be triggered under different conditions, such as a push to the repository or a merge request creation, but it can also be invoked manually by the developer or scheduled to run periodically [24]. Alternatively, or in conjunction with CI/CD, the SAST tool can be integrated into the Integrated development environment (IDE), enabling analysis as soon as the code is written – a preferred method for developers interacting with the program analysis results [22].

Configuration can be centralized and fetched by the tool for each scan or configured separately for each project. The former approach is usually more convenient but requires more effort to set up and maintain, as the configuration must work for all projects using the tool.

While SAST tools typically exhibit a high degree of independence from the analysis environment, integration with the VMS is not always straightforward. As discussed in the next chapter, SAST tools usually include some form of VMS integration, but compatibility with other VMSs may not always be possible.

3.3 Common differences between tools

A wide selection of SAST tools is advantageous but it also implies potential differences between them. This section highlights key aspects to consider as they are likely to significantly impact the user experience in the long term.

3.3.1 Analysis engine

Since code analysis is the primary function of SAST, it is expected that different tools may have unique trade-offs to strike a balance between detection rate and performance.

One such trade-off is the *scope of the analysis*. Some tools may analyze the entire source code of the project to generate highly accurate results while others may only examine the code that has changed since the last scan. The latter approach is usually faster but may miss some vulnerabilities due to the insufficient context.

Another trade-off involves the *supported programming languages*. Designing a tool that supports multiple languages without sacrificing accuracy or performance compared to a single-language tool can be challenging.

Both of these trade-offs share a common characteristic – they are related to the tool's internal code representation, which will be discussed in the next Section 3.3.2.

3.3.2 Analysis engines design

The first step of the static code analysis is to parse the source code into an intermediate representation, so that it can be processed by the analysis engine.

Most of the SAST tools work with the code at the abstraction level of the Abstract Syntax Tree (AST) and Control Flow Graph (CFG), where the syntactic and semantic analysis is performed. As these representations can be easily customized to the needs of the tool, it is not uncommon to see some tools extending the AST with additional metadata or storing their data in a different format, such as a relational database [25]. Another approach is to use the Code Property Graph (CPG), which is a graph representation of the code that combines the information from the AST, CFG and Program Dependence Graph (PDG) [26, 27]. As a next step, the analysis engine is applied to the intermediate representation of the source code.

Analysis types

In static code analysis, several different types of analysis can be performed. The most relevant ones for the SAST tools are the following [28]:

- **Pattern matching** – the most basic analysis that is performed by matching the internal structure with the predefined patterns.
- **Symbolic execution** – the analysis used to detect the execution paths of the code and which inputs can lead to the execution of a particular code block.
- **Program slicing** – the optimization of the symbolic execution where the analysis is performed only on the code that is relevant to the analyzed block.

- **Data flow analysis** – the analysis that is used to detect the possible values of the variables and how they are propagated through the code.
- **Abstract interpretation** – the generalization of the above-mentioned analysis types which allows designing a custom analysis that is tailored to the needs of the tool.

SAST tools use a combination of these analysis types to achieve the desired balance, but the most common approach is to use data-flow analysis and pattern matching. Some of these use cases may even require a ready-to-build source code to be provided [25].

Data-flow analysis may differ between the tools in terms of the following aspects:

- **Scope** – the scope of the analysis may be intra-procedural (within a single function) or inter-procedural (across the whole program).
- **Approximation level** – the level of approximation of the analysis may be a Must analysis (the most precise, where the information is true for all possible executions of the code) or May analysis (the least precise, where the information is true for at least one execution of the code).

3.3.3 Rule language syntax

The rules used by the SAST tools are written in a custom language specific to the tool. Such a language is called a *rule language*, and is expected to be as feature-rich as the tool itself, thus different rule languages can have quite varied syntaxes and semantics. More specific details about the rule languages will be provided later during the analysis of the tools.

3.3.4 Extensibility

Extensibility is a crucial aspect of a SAST tool since it determines the end-users' experience and the extent to which they rely on the tool's developers. One notable aspect of extensibility concerns the common issue of SAST tools lacking support for certain programming languages.

Initially, extending the analysis engine for a new language is required. This can be addressed by the community through creating a plugin if the tool offers the necessary API and documentation for plugin development. In the case of open-source tools, the process can be even more straightforward and transparent, as the community can contribute directly to the tool's codebase. However, even with an extended engine for a new language, the issue of lacking rules for that language persists. If the rule language specification is available, the community can develop rules for the new language with the rule language complexity as the only barrier. Less common are cases where the rule language is not documented but the tool provides an API for creating rules. Each of these approaches has its advantages and disadvantages, but it is important to note that without the ability to utilize user-provided rules, this problem cannot be easily solved at the community level.

3.3.5 Rule maintenance

Proper rule maintenance is essential for a SAST tool to detect recently discovered attack vectors and to ensure that existing rules remain accurate despite new programming language features.

Well-maintained SAST tools are expected to have a large and continuously improving set of rules, reducing false positives and increasing coverage for end-users.

Depending on the tool's extensibility, it may benefit from community contributions, as the community can add support for new languages or keep the rules up-to-date. GitLab, for example, maintains its own version of rules for its GitLab SAST product[4], indicating that SAST tools used within GitLab SAST benefit from enhancements made by such a large organization. This also showcases the increased freedom provided by the tool's extensibility which would be impossible without this level of adaptability.

3.4 Existing state-of-the-art SAST tools

An initial search for existing SAST tools was conducted to identify potential candidates for evaluation. This search resulted in the identification of 24 tools, drawing from academic sources such as studies [9, 25, 29] as well as non-academic sources [30, 31]. Following the initial search, the tools were narrowed down based on the following pre-selection criteria:

[4]https://gitlab.com/gitlab-org/security-products/sast-rules

- **Free** – The tool should be available under a permissive license or be open-source.
- **Coverage** – The tool should provide support for at least 5 programming languages.
- **Maintained** – The tool should be actively maintained, with updates released on a regular basis.

Although limiting the selection to free tools may seem restrictive and may potentially exclude promising candidates, this constraint is part of the Delimitations section and serves as a trade-off to maintain a manageable scope for tool evaluation. The other criteria were considered reasonable as they helped to focus the search on tools that were not abandoned and offered support for multiple programming language families.

Ultimately, 4 tools were chosen for further evaluation based on these criteria. Information about the other considered tools can be found in Appendix A.

- Semgrep – `1.17.0`
- Joern – `1.1.1564`
- SonarQube CE – `9.9.0`
- CodeQL – `2.12.5`

It is important to note that all the information presented about the tools is based on the latest versions available at the time of writing this thesis, as listed above.

3.4.1 Semgrep Open Source (OSS)

Semgrep is an open-source SAST tool that is designed to be as easy to use as possible. Even though the main tool's components and its community-driven rules are distributed under a free license, there are some paid features such as the Semgrep App platform for rules and findings management or DeepSemgrep which extends the Scope by using the Inter-procedural analysis.

3.4.2 Joern

Joern is an open-source code analysis platform that can be used to perform the vulnerability discovery by querying the graph database or to create a custom SAST tool by using the Scala API. Joern uses the custom graph database called OverflowDB to store the code representation.

3.4.3 SonarQube Community Edition

SonarQube provides a platform for code quality management which includes the SAST tool. The tool is available under a free license but the community edition is limited in terms of the number of supported programming languages. While the free version provides an out-of-the-box support only for 19 languages, the paid version supports up to 31 languages. Moreover, the data flow feature that powers taint analysis rules is closed source and not a part of the community edition, therefore many of the advanced security rules are only available in the commercial version of the tool [5].

3.4.4 CodeQL

CodeQL is a proprietary SAST tool designed for security analysis of the code. Besides some license exceptions as usage for open source projects or for research purposes, the tool is only available for the commercial GitHub Advanced Security customers. CodeQL is based on the object-oriented query language called QL. While QL is a closed-source language, its specification is available and the community can contribute new rules to the open-source query library.

3.5 Rule languages

Although the ability to define custom rules was not a requirement for the tools to be included in the evaluation, all of the selected tools provide a

[5] Community Edition doesn't have "Detection of advanced vulnerabilities including Injection Flaws in Java, C#, PHP, Python, JavaScript, TypeScript" [32]

certain level of support for custom rules. More information about the level of support for custom rules can be found in Chapter 4.

Each SAST tool comes with its rule language and corresponding rule syntax that is used to define the rules, which are then used to analyze the source code. The rule languages provided by the tools can be using either a commonly used language or a domain-specific language (DSL). The following sections divide the previously mentioned tools into two groups based on their rule languages:

- **Commonly used languages**

  Semgrep rules are represented in YAML format following the Semgrep rule syntax of pattern-matching expressions.

  SonarQube rules, depending on the analyzed language, can be represented in two different ways. The first way is by creating a custom plugin in Java that extends the SonarQube API. The second way is by using XPath 1.0 expressions to define the rules.

- **Domain-specific language**

  CodeQL queries are represented using the QL language that is optimized for code analysis.

  Joern queries are represented using the patterns in the CPGQL language, which is based on the Scala language and is used to query the generated code property graph.

Chapter 4

Identifying most promising commonly used SAST tools

In this chapter, previously selected tools are compared against each other to determine their suitability for continuous security testing.

The first part of the chapter is dedicated to the comparison of the more general properties of the tools, such as the supported technologies and their integration with the modern CI/CD pipelines. In the second part, we will perform a quantitative analysis of the rulesets of the selected tools and how the rules count has changed over time. Additionally, rules will be classified by their metadata to determine potential gaps in the coverage across different technologies or vulnerability classes. Based on the results of the analysis, the gathered information is then used to identify the tools that will be integrated into the analysis environment automated by the GitLab CI/CD pipeline.

4.1 Technologies coverage

This section provides an overview of the supported technologies by the selected SAST tools. It is mainly based on the official documentation of the tools, but in some cases, we had to use information from the community sources such as the GitHub repositories of the tools or the community-driven projects.

The legend that is used to indicate the degree of support for the technolo-

gies is as follows:
● – Full support for the technology.
○ – Partial support for the technology.
[1] – Custom rules functionality is not supported.
[2] – Functionality requires a third-party plugin.

In the following tables, the technologies are sorted by their popularity according to the StackOverflow 2022 survey [33] based on 53,421 responses from professional developers. Some of the technologies are not included in the tables as they have some serious limitations that make them unsuitable for the majority of the applications[1].

SAST	JavaScript	Python	TypeScript	Java	C#	Bash	PHP	C++	C	Go
Semgrep	●	●	●	●	●	○	●	○	○	●
Joern	○	○	○	○	–	–	–	●	●	–
SonarQube	●[1]	●	●[1]	●	●[1]	–	●	–[2]	–[2]	●[1]
CodeQL	●	●	●	●	●	–	–	●	●	●

Table 4.1: SAST tool's support for the top 10 programming languages sorted by their popularity according to the StackOverflow 2022 survey. Data is based on the official documentation of the tools [32, 34, 35, 36, 37]

Table 4.1 demonstrates that support for the top 10 most popular programming languages is generally quite good, except for the Joern tool. This tool currently does not support 5 out of the 10 languages, resulting in a significant coverage gap. Furthermore, Joern does not claim support for TypeScript, necessitating the compilation of TypeScript code into JavaScript, which complicates the analysis of such code. In contrast to other tools, SonarQube does not fully support C++ and C languages[2]. Semgrep offers basic support for the Bash language, which is not supported by any of the other tools. Analyzing Bash code is particularly crucial for security testing as the language's unique syntax makes it easy to introduce vulnerabilities and is often error-prone [38]. This absence of support for the important language could be because there is already a static analysis tool, called ShellCheck [39], that is dedicated to the analysis of the shell scripts.

[1]Such technologies mostly have very limited support or few to no available rulesets. List of such technologies per tool is as follows:
Semgrep: Closure, Dart, Lisp, Solidity. SonarQube: Kubernetes, Secrets, Text.

[2]SonarQube claims first-class support for C and C++ languages, but this is unavailable in the Community Edition

SAST	Kotlin	Rust	Ruby	Swift	VB.NET	R	Lua	Scala	Elixir	OCaml	Flex
Semgrep	○	○	●	○	–	○	○	●	○	○	–
Joern	○	–	–	–	–	–	–	–	–	–	–
SonarQube	●[1]	–	●[1]	–	●[1]	–	–	●[1]	–	–	●
CodeQL	○	–	●	–	–	–	–	–	–	–	–

Table 4.2: SAST tool's support for the top 11-20 programming languages sorted by their popularity according to the StackOverflow 2022 survey. Data is based on the official documentation of the tools [32, 34, 35, 36, 37]

In contrast, we can immediately observe that the support for the less popular languages in Table 4.2 is much more limited. The only tool that provides at least basic support for more than 3 languages out of the 10 presented is Semgrep. Another interesting observation is the lack of comprehensive support for the Rust language, which has 8.8 % of the respondents in the survey, while Kotlin and Ruby have 9.9 % and 6.7 % respectively, but both are still well supported by the tools.

SAST	HTML	XML	CSS	JSON	YAML	Terraform	Dockerfile	x86/x64	LLVM	JVM
Semgrep	○	○	–	●	○	●	○	–	–	–
Joern	–	–	–	–	–	–	–	●	●	●
SonarQube	●[1]	●	●[1]	–[12]	–[2]	●[1]	–	–	–	–
CodeQL	○	○	–	○	○	–	–	–	–	–

Table 4.3: SAST tool's support for the other technologies. Data is based on the official documentation of the tools [32, 34, 35, 36, 37]

Table 4.3 represents the support for the different technologies that are not programming languages but are still important from a security perspective. For instance, HTML files can affect the security of web applications, while the YAML and JSON file formats are often used to store the configuration of the applications, including the security-related settings. Joern, being primarily a vulnerability discovery and research tool, supports the analysis of the compiled code such as the x86/x64 binaries and JVM bytecode, which makes it unique among many other SAST tools.

4.2 Engine capabilities

Depending on the design and current implementation of the SAST tool's engine, it can be shipped with some extra features that are useful for specific use cases. For instance, most of the tools are able to scan only the whole project, arguing that it is the only way to provide a high-quality analysis. This decision often makes the analysis time much longer, but it does not seem to be a problem for the majority of the users as 57 % of the respondents in the survey stated that they would prefer a slower but more accurate analysis. However, the same survey also showed that 74 % of the respondents are not willing to wait more than a few minutes for the analysis to finish [22]. Consequently, a diff-aware analysis feature, which allows scanning of changed files only, can be a valuable configurable option, particularly in CI/CD pipelines.

Another desirable feature is the ability to scan the codebase without building it, which can be helpful during the development process.

The support for these features among the SAST tools is as follows:

- Semgrep supports scanning changed files only and does not require the code to be buildable.
- Joern does not support scanning only changed files, but it does not require the code to be buildable.
- SonarQube does not support scanning changed files only and varies in its requirement for the code to be buildable, depending on whether the tool is integrated with the build system.
- CodeQL has varying support for scanning changed files only (disabled by default and requires project-specific configuration) and requires the code to be buildable.

4.3 Integration possibilities

As mentioned in Section 3.2 describing different integration methods, there are multiple integration points that should be considered when integrating the SAST tools into the CI/CD pipeline and we will now discuss the support for these integration points in the tools.

4.3.1 Scanning environment

Even though all of the tools provide the CLI interface, the actual analysis is not always performed on the same machine as where the scanning is initiated. Comparing the tools, we can see that the SonarQube is a cloud-based tool that requires the code to be uploaded to the previously deployed SonarQube instance which can be either an on-premise or a cloud-based one. As for the other tools, Semgrep, Joern and CodeQL perform the analysis in the same environment – e.g. the developer's machine – meaning that the code never leaves the local environment.

4.3.2 Cooperation with the VMS

Since 3 out of 4 tools provide an out-of-the-box solution for vulnerability management, we will discuss them separately. Later, the tools will be compared based on the available options for integration with the non-native VMS.

The following list presents the VMS that are shipped with the SAST tools:

- **Semgrep** – can be integrated with the *Semgrep App* – a cloud-based service that provides vulnerability management functionality, which is a part of the community tier with up to 20 users.
- **SonarQube** – has a built-in vulnerability management that is a part of the deployed SonarQube instance.
- **CodeQL** – the vulnerability management is a part of the GitHub Advanced Security that is free for the public GitHub repositories.

In the case of non-native VMS, the relation between the SAST tool and the VMS is rather bidirectional, as the SAST tool is able to export the results to a limited set of formats, which may or may not be supported as an input format by the VMS. In order to solve this compatibility issue, the Static Analysis Results Interchange Format (SARIF), an open standard for the SAST tools to export their results [40], was introduced.

SAST	**Export Formats**	**DefectDojo Import**
Semgrep	SARIF Semgrep JSON JUnit XML	Supported (SARIF, JSON)
Joern	Graph formats	Not supported
SonarQube	HTML API	Supported (HTML, API)
CodeQL	SARIF CSV Graph formats	Supported (SARIF)

Table 4.4: SAST tools' integration capabilities with advanced vulnerability management systems, using DefectDojo as an instance. Referenced from respective tool and DefectDojo documentations [41, 42, 43, 44]

Table 4.4 shows the supported export formats and whether DefectDojo is able to import the results from the corresponding SAST tool. Joern is the only tool that does not support the export of its findings at all, although its intermediate graph representations of code were shown to be useful and are commonly used in the academic research [45, 46, 47]. As for SonarQube, it does not export the results in the SARIF format, which is the standard for the SAST tools. Instead, it only provides an API interface to interact with the SonarQube instance that can be used to retrieve the results. Due to this limitation, the VMS has to be able to interact with the SonarQube API directly or support the import of the HTML reports generated by the *sonar-report* tool [3].

4.4 Metrics of rules quality

Before proceeding to the comparison of the rules quality, we will first discuss the available metrics that can be used during the evaluation. One may assume that all open-source SAST tools have their rules open-sourced as well, but this is not always the case. Both Joern and CodeQL expose every bit of their rules, including the source code, which means that their rules do not introduce any limitations on the analysis. The same can be said about the Semgrep rules, although there are over 350 rules available for the paid users only that were not included in the analysis. On the other hand, SonarQube does not provide the source code of its rules. Instead, only the rule metadata is available[4]. Because the source code for SonarQube's rules is not available,

[3]https://github.com/soprasteria/sonar-report

[4]Except for the infrastructure as code rules: https://github.com/SonarSource/sonar-iac

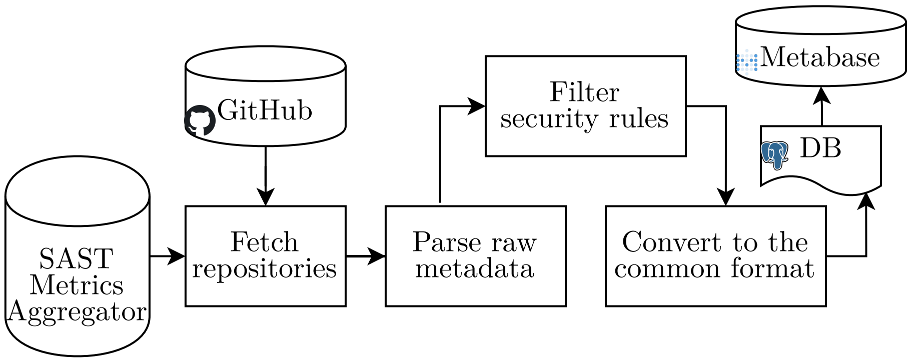


Figure 4.1: SAST Metrics Aggregator architecture.

metrics related to the rules' source code cannot be used for evaluation.

Based on the available data from the official repositories of the tools [48, 49, 50, 51], the following common rule metrics were identified:

- Rule name
- Rule description
- Severity of the rule
- CWE mapping (if applicable)
- Target technology

It is important to note that SAST tools, in addition to the security-related rules, also provide rules for code quality. Although some of the code smells have a significant correlation with security issues [52], the other code smells may not be security-related at all, potentially leading to biased results. For this reason, only the rules that are explicitly marked as security-related were considered in the analysis.

This information can be used to compare the quantity of rules and how they are distributed among languages or CWE categories. However, direct comparison of the rules is not straightforward due to the differences in how the tools structure their rules and rule metadata. To tackle this, the *SAST Metrics Aggregator*[5] was developed. It extracts rule metadata from each tool's official repositories using tailored parsers. This data is then unified into a standard format and stored in a database for further processing. For visual analytics, the open-source *Metabase* Business Intelligence tool is employed [53]. *Metabase* enables customized dashboards and chart creation for analyzing database data. Its configuration can be easily reproduced and deployed on any machine, making it a versatile and convenient tool for subsequent analyses. The *SAST Metrics Aggregator* architecture is depicted in Figure 4.1.

[5]https://github.com/nightshiba/sast-metrics-aggregator

In principle, it should be enough to analyze the commit history of a repository to gain insights into various aspects such as rules quantity and their distribution among identified rule metadata, as well as the evolution of the rules over time. However, in the case of SonarQube, the repository alone does not provide sufficient information for any kind of analysis due to its lack of information regarding the availability of a particular rule in the SonarQube Community Edition or other products. Therefore, to filter out the metadata of unavailable rules, it is necessary to extract information directly from the SonarQube instance through its Web API. This extraction process is semi-automated and should be repeated for each new version of SonarQube. As for the other tools, the data provided by the repositories is sufficient to analyze only the built-in security rules.

Tool	2021-07-02	2021-11-13	2022-04-05	2022-10-18	2023-04-04
Semgrep	*89b246a4*	*6ae3c5f4*	*1de65ed2*	*2c724bed*	*5fe86926*
Joern	*a15933ab*	*ad203d8d*	*8d9c3723*	*bd7e444d*	*601b5c62*
SonarQube	*49aa2dcd*	*620da3ed*	*b9179c27*	*ce950105*	*99d3509f*
CodeQL	*a9c1d3ba*	*b5d37ae0*	*8f3578c9*	*5f39888a*	*f6e774ed*

Table 4.5: Selected time periods for SAST rule metrics analysis and corresponding commit hashes of the official rule repositories [48, 49, 50, 51]

Using the *SAST Metrics Aggregator*, rule metadata was extracted for 5 different versions of each tool (see Table 4.5). The extracted data was then analyzed to answer the following research questions:

- **RQ1:** How many rules are currently available for each tool and how are they distributed among technologies and OWASP Top 10 categories?
- **RQ2:** Are new rules consistently added over time for already supported technologies?
- **RQ3:** Which of the OWASP Top 10 categories are prioritized by each tool?

4.4.1 Rules quantity

To get an overview of the current state of the rules, the data about rules quantity for the most recent time period (2023-04-04) was used. Answering the first research question **RQ1** is the main goal of this section. Due to the differences in how the tools structure their rules and define their metadata, some technologies have to be grouped together to enable a meaningful comparison. Usually, the grouping is applied to the technologies that are similar in nature, such as the *Java* and *Kotlin* languages. However, it should be noted that in some cases, the grouping may be rather misleading, and should be interpreted with caution. For example, the *C* and *C++* languages are grouped together in all tools, even though many rules are much more relevant for only one of them.

The rules distribution among technologies is shown in Figure 4.2. As one might expect, there is a clear correlation between the language popularity and coverage by the tools that support it. Moreover, that correlation may be even stronger than it seems at first glance since the top 5 most popular languages are responsible for more than 60 % of the total number of rules. As for the other languages, the coverage is rather inconsistent and varies greatly between the tools.

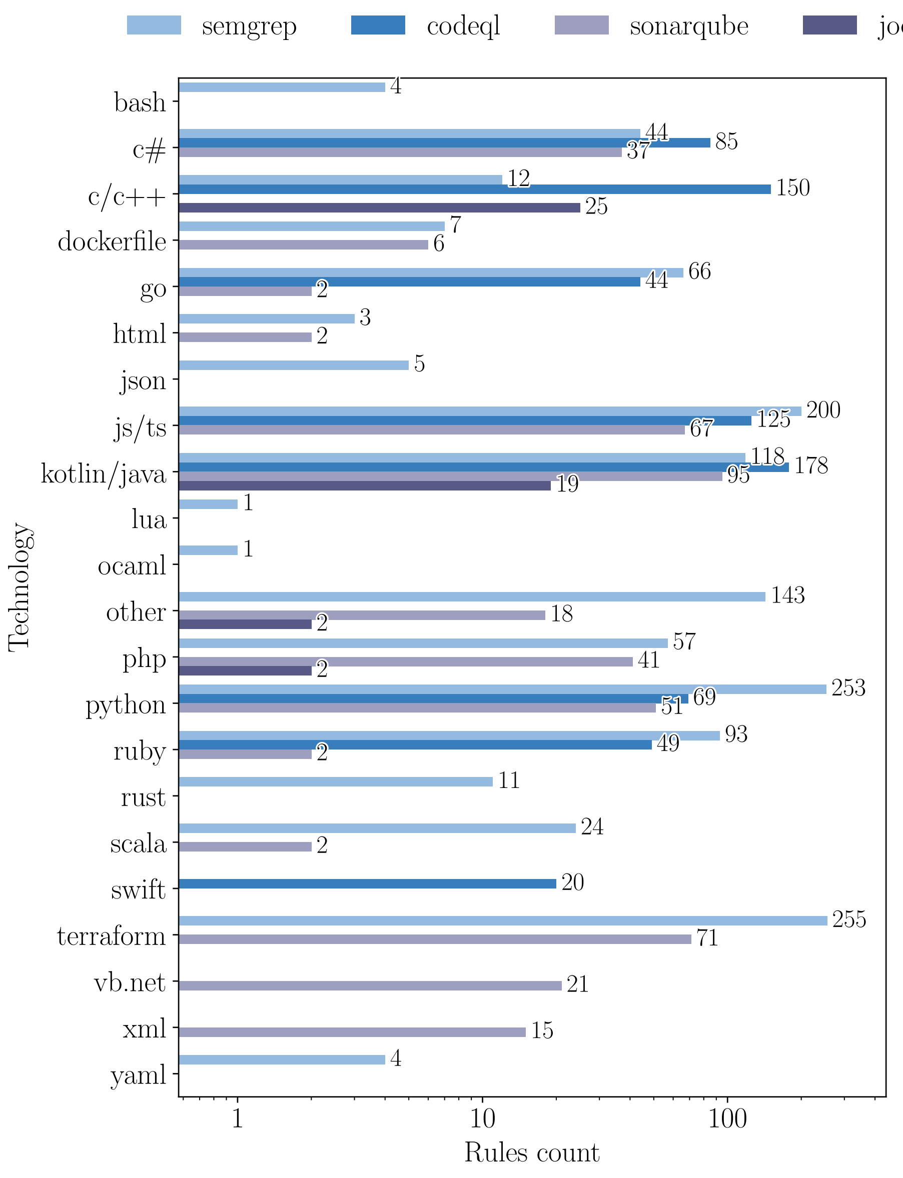


Figure 4.2: Number of security-related rules for each SAST tool and supported technology, some technologies are grouped together.

In Figure 4.3, the rules are grouped by the corresponding OWASP Top 10 category. Unlike the previous chart, the Joern tool was excluded from this one. This is because Joern rules do not have CWE mappings, which are used to determine the OWASP Top 10 category. Interestingly, the CodeQL tool has over 400 rules that are not mapped to the OWASP Top 10 categories. Part of the reason for this is that CodeQL has considerably more rules related to Denial of Service (DoS) attacks than the other tools, such as *CWE-400: Uncontrolled Resource Consumption* and *CWE-730: Denial of Service*.

From the chart, it becomes clear that the tools end up with very similar clusters of rules when it comes to the OWASP Top 10 categories. The *A06: Security Misconfiguration* category is almost empty in all tools, as it is a functionality provided by the specialized SCA tools mentioned earlier in Automated Security Testing. Other categories generally have a similar number of rules in all tools, though there are some exceptions. Most notable one is the fact that the *A03: Injection* and *A10: Server-Side Request Forgery* categories are heavily underrepresented in SonarQube, since the "Injection Flaws" rules are exclusive to the commercial edition of the tool.

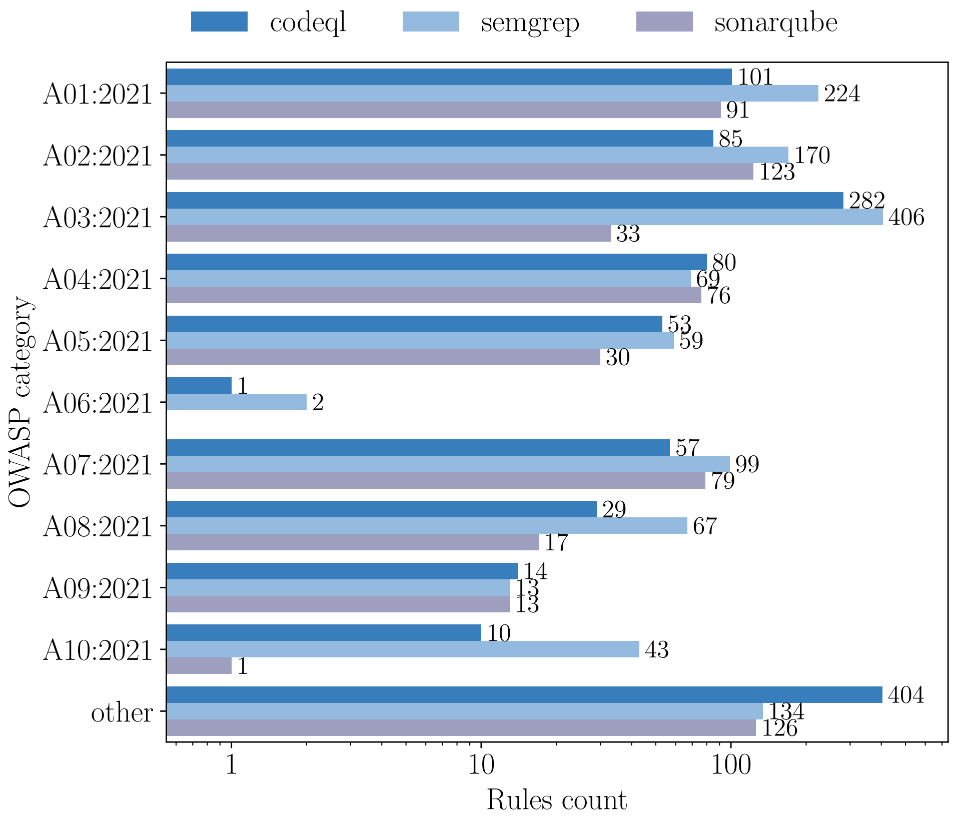


Figure 4.3: Number of security-related rules for each SAST tool and OWASP Top 10 category.

4.4.2 Rule updates frequency

Before answering the second research question **RQ2**, it is important to point out the difference between a rule being added and a rule being updated. In this section, only the rules that were added or removed are considered, while the rules that were updated are ignored due to the lack of data for certain tools.

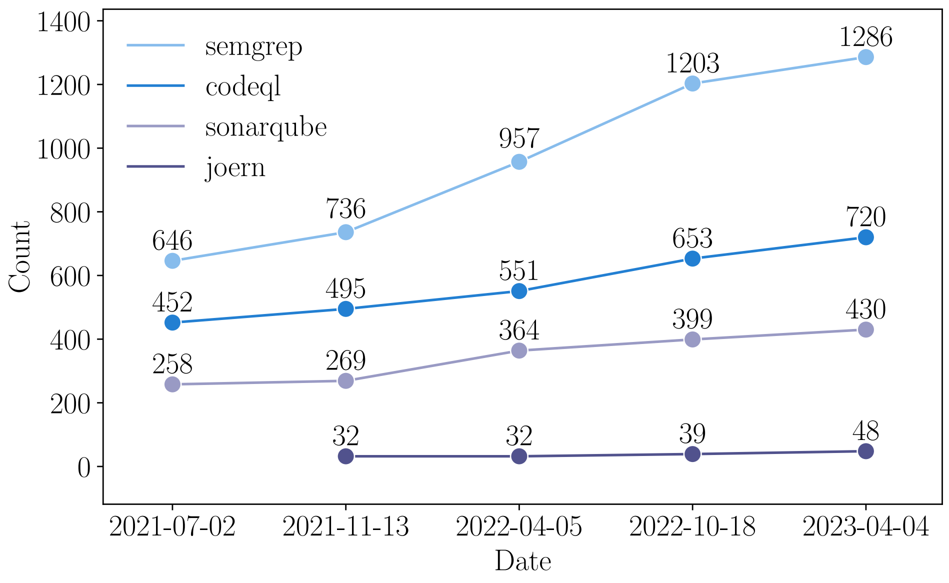
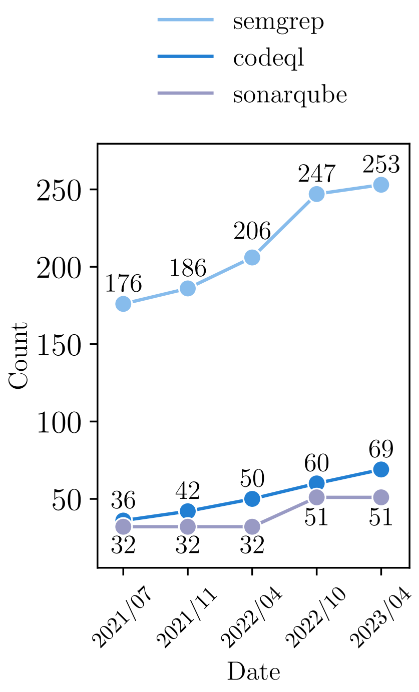


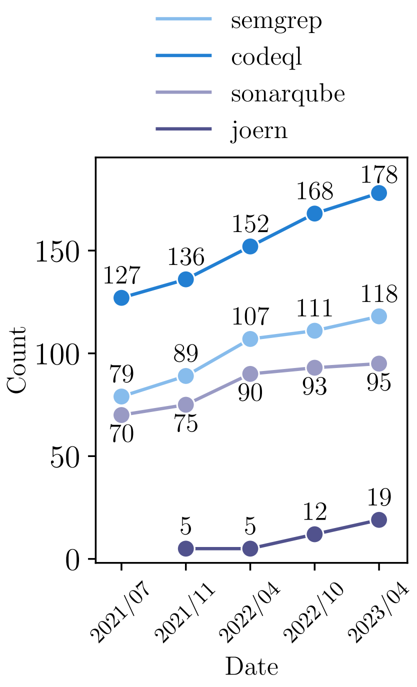
Figure 4.4: Total number of security-related rules history per SAST tool.

Figure 4.4 shows how the security rules quantity has been increasing over time. This chart does not directly answer the research question **RQ2**, but it provides some useful context. Most importantly, it supports the initial assumption that, apart from the iterative updates, the tools can also add a large number of rules at once by porting rules from more specialized security tools. This is especially noticeable for the Semgrep tool which already has over 500 rules ported from 9 different tools[6].

[6]Rules were ported from the following tools: bandit (Python), brakeman (Ruby), eslint (JavaScript/TypeScript), findsecbugs (Java), flawfinder (C/C++), gitleaks (Generic), gosec (Go), phpcs-security-audit (PHP), security-code-scan (C#).



(a) : Python rules history per SAST tool



(b) : Java rules history per SAST tool

Figure 4.5: Number of security-related SAST tool rules for selected languages over time.

Together with the statistics for the specific languages shown in Figure 4.5, it is clear that the tools are not adding new rules for all languages at the same rate. Even though the CodeQL tool was able to maintain a steady growth for all its languages except C#, the other tools were not as consistent. In particular, each of the tools except CodeQL, follows a similar pattern that consists of multiple periods of little to no growth with occasional spikes for the majority of supported languages. Therefore, based on the analyzed data, the answer to the research question **RQ2** is that the tools do not add new rules at a consistent rate.

OWASP	SAST	2021/07	2021/11	2022/04	2022/10	2023/04
A01:2021	codeql	50	**59**	**74**	**90**	**101**
	semgrep	39	**58**	**75**	**213**	224
	sonarqube	31	31	**63**	**80**	**91**
A02:2021	codeql	38	**49**	**59**	**80**	85
	semgrep	83	**97**	**108**	**155**	**170**
	sonarqube	130	108	**127**	128	123
A03:2021	codeql	162	**179**	**207**	**242**	**282**
	semgrep	202	**223**	**294**	**388**	406
	sonarqube	51	34	39	33	33
A04:2021	codeql	43	**53**	**63**	68	**80**
	semgrep	23	**30**	**49**	47	**69**
	sonarqube	20	24	**55**	**66**	**76**
A05:2021	codeql	30	**38**	40	**46**	**53**
	semgrep	21	22	**36**	**59**	59
	sonarqube	28	27	30	30	30
A06:2021	codeql	1	1	1	1	1
	semgrep	–	–	–	2	2
	sonarqube	–	–	–	–	–
A07:2021	codeql	30	**40**	45	**53**	57
	semgrep	19	23	28	**97**	99
	sonarqube	63	59	62	65	**79**
A08:2021	codeql	18	19	24	27	29
	semgrep	35	**41**	**51**	**61**	**67**
	sonarqube	13	14	16	17	17
A09:2021	codeql	4	7	10	12	14
	semgrep	–	–	5	9	13
	sonarqube	16	11	13	13	13
A10:2021	codeql	2	4	8	10	10
	semgrep	24	**35**	40	41	43
	sonarqube	–	1	1	1	1

Table 4.6: Changes in the rules count for the OWASP Top 10 - 2021 categories

Regarding the research question **RQ3**, Table 4.6 depicts the evolution of rule quantity for the OWASP Top 10 categories over time. This table indicates that major increases, shown by growth of 10 % or more, are primarily concentrated in the first several table rows. As anticipated, the tools prioritize OWASP categories that are generally more relevant, implying that higher priority is assigned to the categories at the top of the OWASP Top 10 list.

For the *A02, A05, A07*, and *A09* OWASP categories, it is clear that despite an early lead, SonarQube struggles to maintain pace with other tools and lags behind in terms of rule quantity. The sole notable exception is the *A04* category where the tool managed to double the number of rules in a short time, increasing from 24 rules in 2021/11 to 55 rules in 2022/04. By 2023/04, the rule count has further risen to 76, demonstrating a sustained growth in this category.

4.5 Identifying the tools for integration

All of the presented state-of-the-art SAST tools definitely have much to offer and in many cases can be used as a valuable addition to the security testing process. However, based on the analysis from the previous sections, there are some limitations that affect the integration of the tools into the Software Development Life Cycle (SDLC). The most significant combination of these limitations is present in the Joern, as it the only tool that does not have actively maintained rules and is not able to export the results in a machine-readable format. For this reason, Joern, in its current state, is not suitable to be integrated into the SDLC as a part of the CI/CD pipeline. The other tools have their own limitations, but these are not so restrictive as to prevent or significantly complicate their integration process.

Therefore, only the Semgrep, SonarQube, and CodeQL were selected for further analysis.

Chapter 5

Example SAST project implementation

In this chapter, we will design and implement a template GitLab project called *SAST CI Template* that is capable of running multiple SAST tools using given rules. The aim of this project is to provide a simple interface for evaluating custom rules on the prepared vulnerable codebase, while being a good starting point on how to integrate SAST workflows into the GitLab CI/CD pipeline[1]. Without such an interface, the evaluation of custom rules on the multiple codebases would be a tedious and more error-prone process, as tools differ in their export formats, configuration options and caching mechanisms.
For every ruleset, the project will generate a report containing the findings generated by the corresponding SAST tool and upload it to the DefectDojo instance for further review. This project will be also used to evaluate the custom rules collected during the investigation of rules syntax complexity.

5.1 Project design

To ensure ease of use, the project is expected to abstract away the complexity of tool configuration by providing the configuration files needed to run SAST tools in a stateless environment, so that users can simply clone the project and make use of the available infrastructure to quickly scan the target codebase using the desired rules. Moreover, since the main idea behind the project is not only to use some combination of already existing rules but also to

[1]https://docs.gitlab.com/ee/ci/

evaluate custom rules on the prepared vulnerable codebase, the project has to provide a demo rule written in the syntax of each supported SAST tool. This can serve as a starting point for users to write their own rules and have a consistent workflow for their automatic evaluation, making the development process generally more efficient [54].

As a result, the following requirements for the project have been identified:

R1 Capability to evaluate custom rules on a prepared vulnerable codebase using existing infrastructure[2].
R2 Tool configuration is abstracted by using GitLab CI/CD variables as an interface to workflow configuration.
R3 Observability of the SAST workflow by showing the results of each SAST tool in the GitLab CI/CD pipeline.
R4 Centralized results storage by uploading the generated reports to a DefectDojo instance.
R5 Providing an example of a custom rule for each supported SAST tool.

Overall, the *SAST CI Template* project should minimize the initial effort required yet still be ready for further customization. The project structure is divided into several directories that serve different purposes:

- **src/ci**: contains the source code for the CI/CD pipeline, consisting of GitLab CI job definitions per SAST tool.
- **rules**: contains the custom rules to be used by the SAST tools, including the demo rules and their documentation (as required by **R5**).
- **corpus**: contains the target codebases on which the rules are evaluated.
- **.gitlab-ci.yml**: the GitLab CI/CD pipeline definition that includes the job definitions from **src/ci**.

[2]The project expects the user to have a GitLab instance with a shared runner, a SonarQube instance and a DefectDojo instance. All these instances can be self-hosted or provided as a service.

5.2 GitLab pipeline

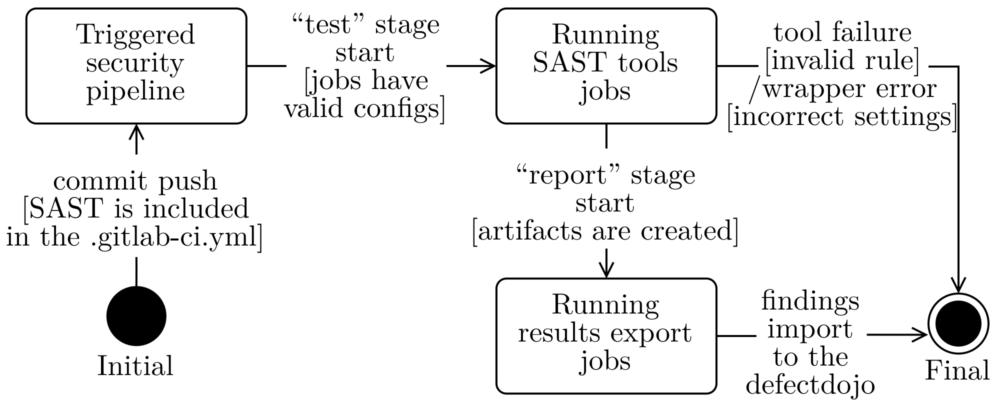

Triggered security pipeline
"test" stage start [jobs have valid configs]
Running SAST tools jobs
tool failure [invalid rule] /wrapper error [incorrect settings]
commit push [SAST is included in the .gitlab-ci.yml]
Initial
"report" stage start [artifacts are created]
Running results export jobs
findings import to the defectdojo
Final

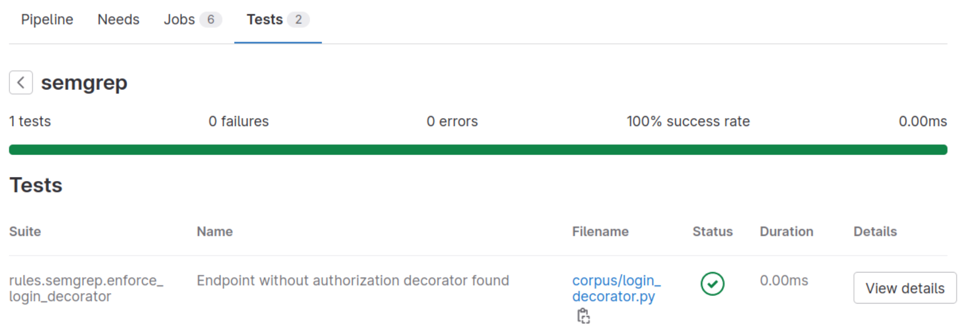
Figure 5.1: SAST CI Template pipeline activity diagram.

The *SAST CI Template* project uses the same approach as the one described in a case study on integrating dynamic security testing tool in CI/CD pipelines [55], where the pipeline is divided into two stages: the testing stage and the evaluation stage. However, in contrast to the case study, the proposed pipeline implements only the first degree of automation named *Continuous Integration (CI)* [3] as it does not include the deployment stage and everything is run in a CI environment.

In this case, the *test* stage is responsible for running the SAST tools whose results are then collected in the *report* stage and uploaded to a DefectDojo instance. Each stage consists of SAST jobs that are executed in parallel, since any job from the *report* stage only depends on the results of the corresponding job from the *test* stage, everything else can be run completely independently. By default, the project is configured to run the SAST pipeline on push events to the default branch and for every merge request. Figure 5.1 shows how SAST tools are integrated into the GitLab CI/CD.

In order to successfully run the SAST pipeline with the provided configuration, the user has only to configure 4 GitLab CI/CD variables, which are used to authenticate the DefectDojo and SonarQube instances. This means that **R2** can be considered fulfilled. More information on how to configure the project can be found in the Appendix B.

[3]https://docs.gitlab.com/ee/ci/introduction/index.html#continuous-integration


Pipeline Needs Jobs 6 Tests 2

semgrep

1 tests 0 failures 0 errors 100% success rate 0.00ms

Tests

Suite	Name	Filename	Status	Duration	Details
rules.semgrep.enforce_login_decorator	Endpoint without authorization decorator found	corpus/login_decorator.py	✓	0.00ms	View details

Figure 5.2: Example of JUnit XML results reported in the GitLab pipeline view.

Upon successful execution of the *test* stage, a JUnit XML file is generated for each SAST tool. This is done by using the *sarif-junit*[4] tool that converts the SARIF reports to the JUnit XML format. The generated JUnit XML files are then processed by GitLab to show the findings directly in the pipeline view, which satisfies **R3**. Figure 5.2 demonstrates how the results are displayed in the GitLab user interface.

5.3 SAST usages

This section describes the implementation details of the designed SAST pipeline. The key difference between running SAST tools in a CI environment and running them locally is that the CI environment usually incorporates some form of virtualization which allows it to execute tasks in a stateless environment and more reliably reproduce the results. In the case of the *SAST CI Template* project, the SAST jobs are executed in a Docker container[5] that is configured to execute the corresponding SAST tool (**R1**) or export its results (**R4**).

5.3.1 CodeQL integration

For the *test* stage of the CodeQL integration, we utilized the *codeql-agent-docker* image from the CodeQL agent project[6], which allows for executing multiple CodeQL commands within a single container. In the *report* stage,

[4]https://github.com/GridexX/sarif-junit
[5]https://docs.docker.com/get-started/#what-is-a-container
[6]https://github.com/codeql-agent-project

the CodeQL SARIF report is uploaded to the DefectDojo instance with the help of the *dd-import* tool[7]. This tool leverages the DefectDojo API to create a new product named *CodeQL-<branch-name>* and imports the findings from the SARIF report which is available as a CI artifact.

While integrating CodeQL, we faced the following challenges:

- **Official Docker image unsuitability.**
  The official *CodeQL Container*[8] provided by Microsoft was initially considered but it proved inefficient for a CI environment due to the need for multiple container invocations and the resulting startup overhead.
- **Query Suite path requirement.**
  Even when using the more suitable *codeql-agent-docker* image, the path to the CodeQL Query Suite was still required. To address this, the job script was configured to automatically detect the appropriate CodeQL Query Suite for the analysis.

5.3.2 Semgrep integration

The Semgrep *test* stage is based on the official Semgrep Docker image[9] which has the Semgrep CLI installed. The *report* stage of the Semgrep integration is almost identical to the already described CodeQL integration, except that the product name has a different prefix.

Only one minor issue was encountered while integrating Semgrep into the pipeline:

- **Semgrep only supports *.yml* rules directory.**
  The Semgrep CLI does not support a rules directory consisting of *.json* and *.yml* files simultaneously. To overcome this limitation, the workaround was implemented to pass the rules one by one using the `-config` option.

[7]https://github.com/MaibornWolff/dd-import
[8]https://github.com/microsoft/codeql-container
[9]https://hub.docker.com/r/returntocorp/semgrep

5.3.3 SonarQube integration

For the *test* stage of the SonarQube integration, the official SonarScanner CLI[10] is used. In the *report* stage, results of the SonarQube analysis are exported to a HTML report using the *sonar-report* tool[11] and uploaded to the DefectDojo instance using the *dd-import* tool.

While integrating SonarQube into the pipeline, the following challenges were encountered:

- **Rule changes require a restart of the SonarQube instance.**
  SonarQube does not support dynamic rule changes since plugins containing the rules are loaded during startup. Although we automated the process of updating the plugins and restarting the SonarQube instance using a helper script, it is still not possible to update the rules without potentially interrupting an ongoing analysis.
- **SonarQube does not support branch analysis.**[12]
  Each SonarQube project is bound to a single branch which means that it is not possible to analyze multiple branches within a single project. To work around this limitation, a SonarQube project is created for each branch beforehand, where the branch name is used as a suffix to the project key. Additionally, the created project is configured to use the equivalently-named quality profile; if it does not exist, the default profile is used instead.
- **HTML report cannot be used to show the findings in the pipeline view.**
  The SonarQube HTML report is not supported by the GitLab pipeline view. In order to show the results in the pipeline view, a conversion script named *sonar-report-sarif* was implemented to convert the HTML report to the SARIF format, which is then used to generate the JUnit XML report.

[10]https://github.com/SonarSource/sonar-scanner-cli

[11]https://github.com/soprasteria/sonar-report

[12]This limitation is present only in the Community Edition of SonarQube. However, there is a plugin available at https://github.com/mc1arke/sonarqube-community-branch-plugin that provides branch analysis support. We did not use this plugin in our evaluation as it would complicate the initial setup.

Chapter 6

Analysis of most promising SAST tools

This chapter investigates the usage of SAST tools, focusing on the practical application and customization of their rule sets. Initially, it assesses the efficiency of default rules in a real-world Python project, analyzing their practicality, strengths and weaknesses. The focus then shifts to the user experience in creating custom rules for these tools, examining the learning curve, potential issues, and the impact of users' security knowledge. The chapter provides insights into the real-world application of SAST tools, the potential for customization and the associated user experience.

6.1 Evaluating default rules in a real-world scenario

In this section, the effectiveness of the default rules[1] provided by SAST tools is assessed by applying them to a real-world project and manually examining the results.

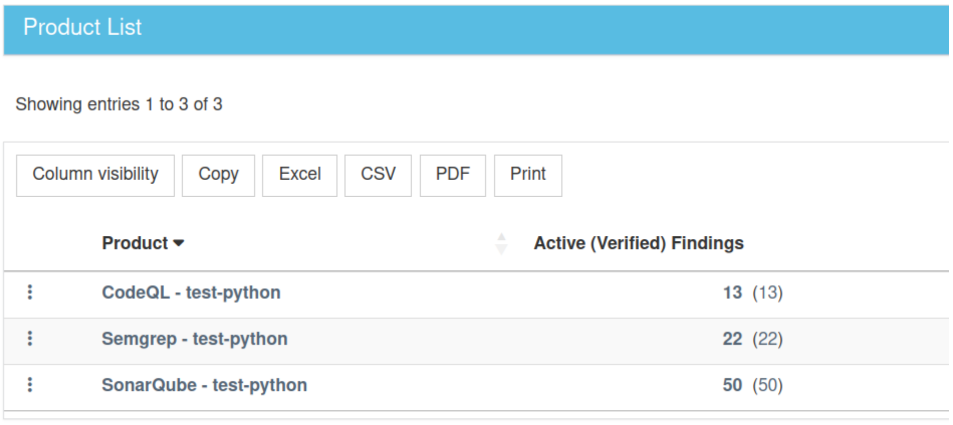
[1]These include the `default` Semgrep ruleset, "Security Hotspots" and "Vulnerabilities" from SonarQube rules, and the `python-security-extended.qls` CodeQL query pack. It should be noted that these were intentionally limited to the Python language.

6.1.1 Choosing the target project

The *deluge* project [2] was chosen as the target for evaluating the default rules. Deluge is a BitTorrent client written in Python featuring multiple user interfaces (UI) such as GTK-UI, Web-UI, and Console-UI. The project is actively maintained, with a codebase of 65 000 lines of code in `.py` files. Furthermore, the project has a history of security issues [56], and there is no evidence of a previous SAST tool usage. These factors make it an ideal candidate for evaluating SAST tools.

6.1.2 Acquiring the results

To analyze the project using the selected SAST tools, the *SAST CI Template* project from Chapter 5 was utilized. Due to its modular design, it was straightforward to add the source code of the *deluge* project and configure the SAST tools to use the default rules with minimal effort. Once the analysis was completed, the results were available in the GitLab UI in the form of reports and uploaded to the DefectDojo instance for further evaluation, as illustrated in Figure 6.1. The SAST analysis artifacts can be found in the corresponding merge request [3].



Product	Active (Verified) Findings
CodeQL - test-python	13 (13)
Semgrep - test-python	22 (22)
SonarQube - test-python	50 (50)

Figure 6.1: DefectDojo findings for the deluge project.

[2]https://github.com/deluge-torrent/deluge
[3]https://gitlab.com/nightshiba1/sast-ci-template/-/merge_requests/2

6.1.3 Analysis of the findings

All the findings from the SAST tools were manually analyzed and categorized into the following groups:

- **Duplicate (DUP)** – The finding is reported multiple times by the same tool.
- **False positive (FP)** – The finding is not a vulnerability or a security issue.
- **Informative finding (IF)** – The finding is a security issue but it cannot be exploited in the current context.
- **Potential vulnerability (PV)** – The finding appears to have a security impact on the application.

The results of the analysis are presented in Table 6.1.

From the analysis, it can be observed that CodeQL had a considerably better false positive rate (only 3 FP) compared to the other tools, as well as unique and relevant informative findings (4 IF) and potential vulnerabilities (4 PV).

In contrast, Semgrep had a higher false positive rate with 16 FP, making it more challenging to distinguish between false positives and actual issues; most findings were related to the same problems and would require rule adjustments for improved precision. It also had 3 IF and only 1 PV.

SonarQube had the worst false positive rate, with numerous duplicate findings (14 DUP) and a high number of false positives (30 FP). Moreover, it reported mostly low severity findings with 6 IF and 0 PV.

Across all SAST tools, there were findings in the testing code or release scripts which could be reduced by configuring the tools to ignore specific files or directories. However, this had minimal impact on the overall number of such findings as there were only a couple of findings per tool at most.

SAST Tool	DUP	FP	IF	PV
CodeQL	2	3	4	4
Semgrep	2	16	3	1
SonarQube	14	30	6	0

Table 6.1: Categorization of the findings in the *deluge* project

6.2 Custom rules usability experiment

In this section, we present the design and execution of a custom rules usability experiment for 3 SAST tools: CodeQL, Semgrep, and SonarQube. These tools have been previously analyzed in depth and the ability to write custom rules for them is of particular importance due to their general-purpose nature. The advantage of these SAST tools lies in the transferable knowledge that allows users to learn how to write custom rules once and then apply this knowledge to other supported languages or technologies. Although an empirical study showed that only 8 % of users write custom rules for program analysis tools and 26 % consider it an important factor [22], the transferability aspect of general-purpose tools makes this experiment highly relevant.

6.2.1 Experiment goals

The experiment has 3 main goals:

- **Learning curve comparison.** Compare the learning curve of the selected SAST tools by assessing the participants' ability to understand and create a basic rule for each tool within a given time frame.
- **Pain points identification.** Identify the pain points experienced by the participants during the experiment when working with each tool.
- **Impact of information security experience.** Investigate if the participants' varying levels of information security experience influence the experiment results.

6.2.2 Experiment setup

To facilitate the experiment, we used the *SAST CI Template* project introduced in Chapter 5 that contains example rule implementations and documentation for each tool. This template project enables participants to set up a development environment for writing custom rules and use the examples as a reference. We also provided a basic code snippet as a target codebase for the rules, instructions for the participants and a questionnaire to be completed after the experiment. Participants were given 4 hours per tool to write a basic rule and the order of the tools was randomized for each participant.

Participants

The selection criteria for the participants included: no prior experience with writing custom rules for SAST tools, familiarity with multiple programming languages (at least Python and Java), and an industry experience as the target audience of this study is developers. Our experiment involved 7 participants with different occupations and levels of information security experience. All of them were familiar with multiple programming languages but had no prior experience with writing custom rules for SAST tools.

#	Occupation	Programming experience	Security background
1	DevOps engineer	6+ years	Moderate
2	Software engineer	4+ years	None
3	Student	5+ years	Strong
4	Software engineer	6+ years	Moderate
5	Machine learning engineer	4+ years	None
6	Software engineer	4+ years	None
7	Student	5+ years	Moderate

Table 6.2: Overview of the participants' background and experience

Tasks

The participants were given 3 tasks, one for each SAST tool, using the same code snippet across all tools to ensure consistency and comparability of the results. Each task required writing a basic rule that could detect a specific vulnerability within the provided code snippet while not flagging the safe portion of the code. Participants were instructed to create simple rules and test them only on this single code snippet, which contained both a minimal working example of a vulnerability and a corresponding safe code portion written in Python programming language. The participants were given 4 hours per tool to write a basic rule, with the restriction that they could not skip a task and move to the next one before completing the current task or the time running out. To verify that the rules written by the participants were indeed successfully implemented and worked correctly, we automatically tested the obtained results using the *SAST CI Template* project.

The following code snippet was used as a target codebase for the rules (some parts were omitted for brevity):

```python
# Example of insecure usage of the python-rq library,
# as the arbitrary user input is passed to the enqueue function
# without any type-checking or sanitization of the parameter.

tasks_redis = Redis.from_url(os.environ.get('REDIS_URL'), db=0)
tasks_queue = Queue('tasks', connection=tasks_redis)

def format_worker(text):
    return f'Formatted text: {text}'

def safe_testformat(request):
    text = request.GET.get('text', '')
    tasks_queue.enqueue(format_worker, 'test')
    # ...
    return HttpResponse(f'Result: {result}')

def unsafe_format(request):
    text = request.GET.get('text', '')
    tasks_queue.enqueue(format_worker, text)
    # ...
    return HttpResponse(f'Result: {result}')
```

Procedure

The experiment procedure involved the following steps:

1. Understand the vulnerability and the safe code snippets.
2. Set up a development environment for writing custom rules using the provided template project for the assigned tool.
3. Test the template project by verifying that the example rule implementation works correctly.
4. Write a basic rule that can detect the vulnerability in the given code snippet.
5. Test the rule to ensure it works correctly.
6. Complete the questionnaire.
7. Repeat the steps for the next tool until all tools are completed or the time runs out.

The questionnaire collected information on time spent on each task, success or failure in completing the task, reasons for failure, and pain points experienced during the experiment. Additionally, two open questions were included to capture participants' insights:

- Participants were asked about their overall experience working with the 3 SAST tools and how they felt about the custom rules writing process: *How satisfied are you with the custom rule writing experience for each of the SAST tools?*
- Participants were asked whether they believed it was a good idea for developers to write custom rules for their projects rather than relying on predefined rules: *Do you imagine a scenario where Software Engineers write custom rules for their projects to prevent vulnerabilities or enforce best practices?*

The experiment procedure and questionnaire were distributed to the participants as part of an archive which can be found at the provided link [4].

[4]https://gitlab.com/nightshiba1/sast-ci-template/-/releases/sast-complexity-experiment

6.2.3 Results

#	Task 1	Task 2	Task 3
1	✓CodeQL (1.5h)	✓SonarQube (1.5h)	✓Semgrep (0.1h)
2	✓CodeQL (3.5h)	✓Semgrep (1h)	– SonarQube (4h)
3	✓SonarQube (3.5h)	✓CodeQL (2.2h)	✓Semgrep (0.25h)
4	✓SonarQube (0.6h)	✓Semgrep (0.15h)	✓CodeQL (0.5h)
5	✓Semgrep (1.25h)	✓CodeQL (2.5h)	– SonarQube (4h)
6	✓Semgrep (2h)	– SonarQube (4h)	– CodeQL (4h)
7	✓Semgrep (0.25h)	✓CodeQL (2.7h)	✓SonarQube (2.5h)

Table 6.3: Overview of the participants' performance in the experiment

The results of the experiment were obtained from the completed questionnaires and metadata, including the source code of the rules written by the participants. To analyze the data, we focused on addressing the goals of the experiment.

Learning curve comparison

Semgrep seems to have the simplest syntax of the three tools as all participants were able to complete the task successfully and spent the least amount of time on it, taking an hour or less on average. Moreover, the participants reported that Semgrep had the most intuitive syntax and was the easiest to start with. A total of 5 participants also mentioned that they appreciated Semgrep's documentation and the speed of the tool.

Conversely, the participants spent the most time on SonarQube and only 4 out of 7 participants were able to complete the task successfully. Since all participants are familiar with the Java programming language, none of them reported having issues with the language itself. However, every participant reported experiencing a complicated development environment setup and a significant lack of documentation which added to the difficulties encountered at the very beginning of their engagement with the tool.

CodeQL, although better in terms of documentation and development environment setup compared to SonarQube, has the most complex syntax among the SAST tools due to its advanced dataflow capabilities. Most participants were able to complete the task successfully but the time spent was significantly higher than for Semgrep. Additionally, all participants reported that the tool has a steep learning curve due to the various concepts of the QL language.

Pain points identification

For Semgrep, there were no pain points that were reported by multiple people, indicating that participants had a relatively smooth experience with the tool.

In CodeQL, 3 participants reported that they had occasional problems with the IDE, requiring them to restart VSCode during the experiment due to some CodeQL extension errors. Among the participants, 5 mentioned that, despite CodeQL having detailed documentation, they found it challenging to understand the concepts and apply them effectively without frequently resorting to the Query Results and AST viewer for guidance. Furthermore, 2 participants expressed dissatisfaction with the available debugging tools being slow due to the recompilation of the query after each change.

In SonarQube, all 7 participants reported that there is a significant lack of proper documentation for writing custom rules and this issue notably slowed them down. Every participant who successfully finished the task eventually had to debug the SonarQube API runtime to overcome some of the issues they encountered.

Impact of overall experience

The participants with more extensive security and programming experience tended to achieve better results, completing tasks faster and more successfully. Still, it remains uncertain whether these outcomes directly correlate with their security background, programming skills, or both. Factors like proficiency in debugging new tools or a strong motivation to learn about security could also affect their performance.

In terms of experience with the three SAST tools and custom rule writing, many participants favored the idea of writing project-specific rules to prevent vulnerabilities or enforce best practices, instead of relying on preset rules. However, they also expressed a preference for having a specialist handle advanced custom rule writing, as it might be too complex for developers lacking the required expertise. This implies that while developers see the importance of custom rules, they might prefer security experts to be involved to ensure effective and efficient implementation.

6.3 User recommendations

Taking into account the data gathered throughout this research, it can be concluded that general-purpose Static Application Security Testing (SAST) tools can efficiently detect common vulnerabilities and provide a reasonable degree of security, especially in large-scale projects.

For instance, when dealing with open-source projects on GitHub or organizations using GitHub Advanced Security, CodeQL appears to be a good choice. It offers high precision and integrates well within the GitHub environment.

In contrast, Semgrep might be a better pick in other situations due to its simple use, speed, and the fact that it is open-source, giving users more control over the tool.

On the other hand, SonarQube Community Edition, due to its limited features compared to its commercial versions, seems more suitable as a code quality tool with some added security features, rather than a full-blown SAST tool.

Nonetheless, it's important to remember that even the best general-purpose SAST tools have their limitations. Depending on the security goals of a project or organization, language-specific SAST tools may be more suitable if no constraints conflict with this choice.

Custom rules can also be beneficial, particularly for organizations that wish to leverage the scalability of general-purpose SAST tools, or for individual projects targeting specific vulnerabilities. However, the trade-offs of using custom rules, such as the time and effort needed to write and possibly maintain them, should be carefully considered.

Chapter 7

Conclusion

In this thesis, a comprehensive examination of Static Application Security Testing (SAST) tools and their integration into the Software Development Life Cycle (SDLC) is conducted, aiming to provide the essential knowledge needed to make well-informed decisions regarding SAST tool adoption and integration. A thorough comparison and assessment led to the identification of CodeQL, Semgrep, and SonarQube as top-performing SAST tools, each with distinct functionalities. The rule analysis framework, developed and utilized during this process, facilitated an automatic analysis of the rules each tool offers.

A practical SAST project implementation using GitLab is presented, providing a useful template for those looking to evaluate custom rules and integrate multiple SAST tools into their development process. Furthermore, an evaluation of SAST tools in a real-world scenario, using default rules, provided insights into how these rules influence the findings produced. This evaluation highlighted the precision of the tools, the uniqueness of their findings, and their tendency to generate false positives.

Additionally, the custom rules usability experiment offered valuable information on the learning curves, challenges, and influence of experience associated with each SAST tool, further assisting in the selection of the most suitable tool based on specific needs.

In conclusion, this thesis delivers practical insights that contribute to the improvement of application security by facilitating the effective adoption and integration of SAST tools, as well as highlighting the importance of vulnerability detection and prevention throughout the development process.

7.1 Goals fulfillment

The main objective of this thesis was to evaluate general-purpose SAST tools and provide insights to help developers and organizations choose the most suitable tool based on their requirements and resources. In order to achieve this objective, we addressed the specific goals outlined in the thesis assignment.

Goal 1: *Explain the main considerations of security of software – threats, vulnerabilities, modeling, evaluation.*

We explored the primary aspects of software security in Chapter 2, discussing threats, vulnerabilities, modeling and evaluation. This included an overview of the phenomenon of vulnerabilities, vulnerability scoring systems and various vulnerability types.

Goal 2: *Discuss theoretical approaches to automated solutions to the problems of software security. Focus particularly on the concept of static application security testing (SAST), including the integration of SAST tools in a project's Continuous Integration/Continuous Delivery (CI/CD) pipeline.*

Theoretical approaches to automated solutions for software security were covered in Chapters 2 and 3, with a special emphasis on SAST and its integration into a project's CI/CD pipeline. We explained the differences between manual and automated security testing and provided an overview of the key components of the SAST workflow and its integration into the SDLC.

Goal 3: *Research the current functionality and limitations of commonly used general purpose SAST tools.*

We investigated the features and constraints of widely used general-purpose SAST tools in Chapter 3 and 4. This included exploring the variations between tools, such as analysis engine design, extensibility, rule language syntax and rule maintenance. We compared 4 state-of-the-art SAST tools (CodeQL, Semgrep, SonarQube and Joern) and selected the most promising ones for integration into the SDLC, based on their technologies coverage, engine capabilities, integration possibilities and metrics of the available security rules. In order to perform the rule analysis, we introduced the "SAST Rules Aggregator" in Section 4.4, which is a modular rule analysis framework that provides a unified interface for comparing and analyzing rules from multiple SAST tools.

Goal 4: *Design and implement a testing environment (preferably using GitLab CI or other open repository) that could be used to conduct a comparison of the selected tools. Use the implemented environment to analyze the suitability of the researched tools for the purpose of continuous security testing of applications throughout their development lifecycle.*

In Chapter 5, we designed and implemented a GitLab CI testing environment called "SAST CI Template" to compare the selected tools. This environment facilitated the evaluation of custom rules on a target codebase and showcased the use of GitLab CI/CD pipelines for SAST workflows. The results yielded valuable insights into the integration process of each tool and highlighted the main challenges associated with

SAST workflow integration. We leveraged this environment in Chapter 6 to assess the researched tools' suitability for continuous security testing of applications throughout their development lifecycle, by applying the tools to find vulnerabilities in real-world projects and conducting a custom rules usability experiment.

Goal 5: *Discuss your results, provide recommendations to users.*

In Chapter 6, we discussed the outcomes of the evaluation of default rules presented in Section 6.1.3, which provided insights into the potential SAST results. This assessment was complemented by an in-depth experiment discussed in Section 6.2.3. Based on these investigations, we proposed recommendations aimed at guiding developers and security teams in the selection of the most suitable SAST tool to align with their specific needs and resources.

Overall, this thesis successfully achieved its objectives by providing a detailed analysis of SAST tools, their integration methods, and their potential impact on software security. The results and insights derived from this research can serve as a valuable resource for developers, organizations, and security experts aiming to enhance their software's security through the adoption of SAST tools.

7.2 Future work

In future research, a similar approach could be applied to analyze and compare commercial versions of SAST tools, which were not included in this thesis. A more in-depth analysis could also be conducted by comparing the support of specific programming language libraries or programming language-specific SAST engine features across different tools. Moreover, there is potential for enhancing the rule analysis framework by introducing a module that automates the process of identifying missing rules in a given SAST tool that are present in other tools, enabling more efficient rule portability.

Appendix A

Other Considered SAST Tools

This appendix provides a brief overview of the tools that were considered but not selected for further evaluation, and the reasons for their exclusion.

- **Specialized on a particular language family**:
  - Progpilot
  - RIPS
  - PHPsafe
  - WAP
  - Bandit
  - Flawfinder
  - PVS-Studio
  - Brakeman
- **Not actively maintained or updated**:
  - Agnitio
  - FindBugs
  - OWASP Code Crawler
  - Find Security Bugs
  - clj-holmes
- **Commercial (limited quotas or trial periods)**:
  - Checkmarx
  - Codacy
  - Coverity
  - Fortify
  - SpotBugs
  - Veracode
  - Snyk Code

Appendix B

Guide for setting up SAST CI Template

In order to use the SAST CI Template, the user must be logged into the GitLab instance that has at least one shared runner available. Additionally, the user must have a SonarQube instance and a DefectDojo instance available, along with the API keys for both of them.

Variable	**Description**
DD_URL	URL of the DefectDojo instance.
DD_API_KEY	API key for the DefectDojo user.
SONARQUBE_URL	URL of the SonarQube instance.
SONARQUBE_TOKEN	API token for the SonarQube user.

Table B.1: Required GitLab CI/CD variables for SAST CI Template.

1. Create a new project in GitLab by cloning the `sast-ci-template` repository.
2. Go to the project's Settings → CI/CD → Variables.
3. Add the required variables (see Table B.1).
4. Go to the project's CI/CD → Pipelines.
5. Click on the `Run pipeline` button.
6. Wait for the pipeline to finish and check the results in DefectDojo.

Appendix C

Bibliography

[1] Raul L Katz and Pantelis Koutroumpis. Measuring digitization: A growth and welfare multiplier. *Technovation*, 33(10-11):314–319, 2013.

[2] Krzysztof Cabaj, Dulce Domingos, Zbigniew Kotulski, and Ana Respício. Cybersecurity education: Evolution of the discipline and analysis of master programs. *Computers & Security*, 75:24–35, 2018.

[3] Max Smeets. A matter of time: On the transitory nature of cyberweapons. *Journal of Strategic Studies*, 41(1-2):6–32, 2018.

[4] Hossein Keramati and Seyed-Hassan Mirian-Hosseinabadi. Integrating software development security activities with agile methodologies. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 749–754. IEEE, 2008.

[5] Radek Fujdiak, Petr Mlynek, Pavel Mrnustik, Maros Barabas, Petr Blazek, Filip Borcik, and Jiri Misurec. Managing the secure software development. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–4. IEEE, 2019.

[6] Paul E Black, Barbara Guttman, and Vadim Okun. Guidelines on minimum standards for developer verification of software. *arXiv preprint arXiv:2107.12850*, 2021.

[7] Achim Brucker and Uwe Sodan. Deploying static application security testing on a large scale. *Sicherheit 2014–Sicherheit, Schutz und Zuverlässigkeit*, 2014.

[8] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[9] Sadeq Gholami and Zeineb Amri. Automated secure code review for web-applications, 2021.

[10] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth symposium on usable privacy and security (SOUPS 2018)*, pages 281–296, 2018.

[11] Michael Whitney, Heather Lipford-Richter, Bill Chu, and Jun Zhu. Embedding secure coding instruction into the ide: A field study in an advanced cs course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 60–65, 2015.

[12] Jonathan M Spring, Allen Householder, Eric Hatleback, Art Manion, Madison Oliver, Vijay Sarvapalli, Laurie Tyzenhaus, and Charles Yarbrough. Prioritizing vulnerability response: A stakeholder-specific vulnerability categorization (version 2.0). Technical report, Carnegie Mellon University, 2021.

[13] Common vulnerability scoring system (cvss). `https://www.first.org/cvss/v3.1/specification-document`. Accessed: 2022-09-30.

[14] The open web application security project® (owasp). `https://owasp.org/Top10/`. Accessed: 2022-09-30.

[15] Matthew Bach-Nutman. Understanding the top 10 owasp vulnerabilities. *arXiv preprint arXiv:2012.09960*, 2020.

[16] Gaoqi Liang, Junhua Zhao, Fengji Luo, Steven R Weller, and Zhao Yang Dong. A review of false data injection attacks against modern power systems. *IEEE Transactions on Smart Grid*, 8(4):1630–1638, 2016.

[17] M Hassan, M Ali, T Bhuiyan, M Sharif, and S Biswas. Quantitative assessment on broken access control vulnerability in web applications. In *International Conference on Cyber Security and Computer Science 2018*, 2018.

[18] John Steven. Threat modeling-perhaps it's time. *IEEE Security & Privacy*, 8(3):83–86, 2010.

[19] Michael A Howard. A process for performing security code reviews. *IEEE Security & privacy*, 4(4):74–79, 2006.

[20] Francesc Mateo Tudela, Juan-Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan-Antonio Sicilia Montalvo, and Michael I Argyros. On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications. *Applied Sciences*, 10(24):9119, 2020.

[21] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106. IEEE, 2011.

[22] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[23] Linghui Luo, Martin Schäf, Daniel Sanchez, and Eric Bodden. Ide support for cloud-based static analyses. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, pages 1178–1189, 2021.

[24] Charanjot Singh, Nikita Seth Gaba, Manjot Kaur, and Bhavleen Kaur. Comparison of different ci/cd tools integrated with cloud platform. In *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 7–12. IEEE, 2019.

[25] Ashot A Grigorian, Slinger Jansen, and Gerard Wagenaar. A competition analysis of software assurance tools. *Secureseco*, 2022.

[26] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[27] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[28] Jiří Slabý. Automatic bug-finding techniques for large software projects. *PhD's thesis, Masaryk University, Marec*, 2013.

[29] Feras Al Kassar, Giulia Clerici, Luca Compagna, Fabian Yamaguchi, and Davide Balzarotti. Testability tarpits: the impact of code patterns on the security testing of web applications. 2023.

[30] Compatibility program: Assessment and remediation tools. `https://cwe.mitre.org/compatible/category.html#Assessment%20and%20Remediation%20Tool`. Accessed: 2022-11-05.

[31] Source code analysis tools. `https://owasp.org/www-community/Source_Code_Analysis_Tools`. Accessed: 2022-11-01.

[32] Sonarqube downloads page. `https://www.sonarsource.com/products/sonarqube/downloads/`. Accessed: 2023-03-10.

[33] Stack overflow developer survey 2022. `https://survey.stackoverflow.co/2022/#most-popular-technologies-language-prof`. Accessed: 2023-01-10.

[34] Codeql supported languages. `https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/`. Accessed: 2023-03-10.

[35] Joern supported languages. `https://docs.joern.io/home#supported-languages`. Accessed: 2023-03-10.

[36] Semgrep supported languages. `https://semgrep.dev/docs/supported-languages/`. Accessed: 2023-03-10.

[37] Sonarqube supported languages for custom rules. `https://docs.sonarqube.org/latest/extension-guide/adding-coding-rules/`. Accessed: 2023-03-10.

[38] Zheyang Li. An empirical study on bash language usage in github. Master's thesis, University of Waterloo, 2021.

[39] Shellcheck - a shell script static analysis tool. `https://github.com/koalaman/shellcheck`. Accessed: 2023-01-12.

[40] Oasis static analysis results interchange format (sarif). `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif`. Accessed: 2023-01-13.

[41] Codeql export formats. `https://docs.github.com/en/code-security/codeql-cli/codeql-cli-manual/bqrs-interpret#primary-options`. Accessed: 2023-04-01.

[42] Joern export formats. `https://docs.joern.io/exporting`. Accessed: 2023-04-01.

[43] Semgrep export formats. `https://semgrep.dev/docs/cli-reference/#semgrep-scan-command-options`. Accessed: 2023-04-01.

[44] Defectdojo supported import formats. `https://defectdojo.github.io/django-DefectDojo/integrations/parsers/`. Accessed: 2023-04-01.

[45] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[46] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016.

[47] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[48] Codeql libraries and queries. `https://github.com/github/codeql`. Accessed: 2023-02-01.

[49] Joern query database. `https://github.com/joernio/joern/tree/master/querydb`. Accessed: 2023-02-01.

[50] Semgrep rules registry. `https://github.com/returntocorp/semgrep-rules`. Accessed: 2023-02-01.

[51] Sonarsource rules specification repository. `https://github.com/SonarSource/rspec`. Accessed: 2023-02-01.

[52] Kazi Zakia Sultana, Zadia Codabux, and Byron Williams. Examining the relationship of code and architectural smells with software vulnerabilities. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40. IEEE, 2020.

[53] Bruno Santos, Francisco Sério, Steven Abrantes, Filipe Sá, Jorge Loureiro, Cristina Wanzeller, and Pedro Martins. Open source business intelligence tools: Metabase and redash. In *KDIR*, pages 467–474, 2019.

[54] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482. IEEE, 2021.

[55] Thorsten Rangnau, Remco v Buijtenen, Frank Fransen, and Fatih Turkmen. Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 145–154. IEEE, 2020.

[56] Deluge-torrent disclosed vulnerabilities. `https://www.cvedetails.com/vulnerability-list/vendor_id-16504/Deluge-torrent.html`. Accessed: 2023-04-22.

Appendix D

Attachments

```
├── readme.txt ................... Brief description of the attachments
├── metrics.zip ... Archive of the SAST Metrics Aggregator repository[1]
├── template.zip ........ Archive of the SAST CI Template repository[2]
├── artifacts.zip ................. Findings from the rules evaluation[3]
├── experiment ........... Information collected during the experiment
│   ├── assignment.zip .......... Assignment given to the participants[4]
│   ├── rules ......................... Rules written by the participants
│   └── survey ............................... Questionnaire responses
└── text .................................... Source code of the thesis
    ├── thesis.pdf ............................ Thesis in PDF format
    └── thesis_assignment.pdf ...... Thesis assignment in PDF format
```

[1]https://github.com/nightshiba/sast-metrics-aggregator

[2]https://gitlab.com/nightshiba1/sast-ci-template

[3]https://gitlab.com/nightshiba1/sast-ci-template/-/merge_requests/2

[4]https://gitlab.com/nightshiba1/sast-ci-template/-/releases/sast-complexity-experiment

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Golovyrin Leonid**	Personal ID number:	**498960**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Analysis of tools for static security testing of applications

Bachelor’s thesis title in Czech:

Analýza nástrojů pro statické testování bezpečnosti aplikací

Guidelines:

1) Explain the main considerations of security of software - threats, vulnerabilities, modeling, evaluation.
2) Discuss theoretical approaches to automated solutions to the problems of software security. Focus particularly on the concept of static application security testing (SAST), including the integration of SAST tools in a project's Continuous Integration/Continuous Delivery (CI/CD) pipeline.
3) Research the current functionality and limitations of commonly used general purpose SAST tools.
4) Design and implement using GitLab CI a testing environment (preferably using GitLab CI or other open repository) that could be used to conduct a comparison of the selected tools.
5) Implement the designed environment and use it to analyze the suitability of the researched tools for the purpose of continuous security testing of applications throughout their development lifecycle.
6) Discuss your results, provide recommendations to users.

Bibliography / sources:

Salter, C., Saydjari, O. S., Schneier, B., Wallner, J.: Toward A Secure System Engineering Methodology. Proceedings of the 1998 workshop on New security paradigms. 1998.
Shostack, A.: Threat Modeling: Designing for Security. Wiley, 2014, 9781118809990.
Howard, M., LeBlanc, D.: Writing Secure Code, 2nd Edition. Microsoft Press, 2003, 9780735617223.
Chess, B., West, J.: Secure Programming with Static Analysis. Addison-Wesley. 2007. ISBN 978-0-321-42477-8.

Name and workplace of bachelor’s thesis supervisor:

Ing. Josef Kokeš   Department of Information Security FIT

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **30.08.2022**   Deadline for bachelor thesis submission: __________

Assignment valid until: **19.02.2024**

Ing. Josef Kokeš Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature