Insights from Running 24 Static Analysis Tools on Open Source Software Repositories

Fabiha Hashmat, Zeyad Alwaleed Aljaali, Mingjie Shen, and Aravind Machiry

Purdue University, West Lafayette, IN, USA
{fhashmat,zaljaali,shen497,amachiry}@purdue.edu

Abstract. OSS is important and useful. We want to ensure that it is of high quality and has no security issues. Static analysis tools provide easy-to-use and application-independent mechanisms to assess various aspects of a given code. Many effective open-source static analysis tools exist. In this paper, we perform the first comprehensive analysis using 24 open-source static analysis tools (through Omega Analyzer) on 4,947 repositories. Our study identified several interesting findings, such as the distribution of errors in relation to the criticality score of repositories shows that repositories with a criticality score have the highest percentage of errors. We envision that our findings provide insights into the effectiveness of static analysis tools on OSS and future research directions in securing OSS repositories.

Keywords: Program Analysis · Omega Analyzer· GitHub Network Projects · Static Analysis · OSSF critical repositories

1 Introduction

Open Source Software (OSS) plays and an important role in the software ecosystem [18, 70]. Many important and high-impact software products, such as Linux kernel [69] and nodejs [8], are all open source. It is also well-known that many organizations use OSS as part of their software products [36]. Given the prevalence, it is important to ensure that OSS follows good engineering practices to avoid security vulnerabilities. Furthermore, WhiteHouse recently released an official report [7] emphasizing the importance of securing OSS and the use of secure software engineering practices.

Static Application Security Testing (SAST) [28, 31] is a well-known practice to detect common vulnerabilities. However, previous studies [66] have shown that 40% of organizations do not use any SAST tools as part of their engineering practices. Furthermore, our analysis (in § 2), also shows that only 19% of the critical OSS use any SAST tools. There are various Free SAST Tools (FSTs), such as CodeQL [2], to detect common security vulnerabilities and can be used without cost or licensing issues. However, the effectiveness of FSTs on OSS is unknown. Although prior works (discussed in § 8.2) try to explore the same aspect, they mainly focus on SAST tools that find software vulnerabilities. Furthermore, their study is often limited to a small number of OSS projects.

In this paper, we perform the first large-scale study of the effectiveness of FSTs on OSS. Specifically, we investigate the following three research questions:

- **RQ 1: Execution.** *Executing FSTs on OSS Repositories.* How easy/hard is it to run static analysis tools on OSS repositories? How robust are the tools in analyzing these repositories?
- **RQ 2: Effectiveness.** *Effectiveness of FSTs on OSS Repositories.* What types of issues are found by FSTs? What types of issues are prevalent in OSS repositories?
- **RQ 3: Quality.** *Quality of Issues Found by FSTs.* What is the quality (true/false positive rate) of the issues found FSTs?

For our dataset, we collected a suite of 24 FSTs from Open Source Security Foundation (OpenSSF)'s [3] AlphaOmega [1] project. The project contains a list of stable and recommended SAST tools for OSS projects. We also identified a set of 4,947 critical OSS projects by contacting OpenSSF team. We created a GitHub workflow [4] (i.e., a Continuous Integration (CI) pipeline) called FSTWorkflow, that can execute all 24 FSTs on a given repository. We also created an automated mechanism to categorize the tool's results in a common format. We investigated our research questions by executing FSTs (through FSTWorkflow) on all of our repositories and analyzed the results through automated and manual analysis. In summary, the following are our contributions:

- **(FSTs Collection.)** We collected a dataset of 4,947 repositories across various programming languages to evaluate Omega Analyzer's performance.
- **(Study.)** We performed the first extensive analysis of Omega Analyzer, identifying resource constraints, failure modes, and language distribution.
- **(Findings.)** We found a 98.3% success rate for Omega Analyzer, with Python, JSON, and JavaScript being the most common languages. The Source Code Scanning (SCS) tool detected more errors and warnings than the Misconfigurations (MC) tool.
- **(Open-source availability.)** We made our analysis workflow, dataset, and results open-source to enable future research.

2 Motivation

OSS is important and used directly or indirectly in many software products [22]. The design and code patterns used in OSS also inspire other software products [53]. It is important to ensure that OSS does not contain any obvious security vulnerabilities in general insecure (or risky) practices, e.g., unsanitized use of `strcpy`, hardcoded private keys, etc.

SAST is a recommended practice to easily detect previously known vulnerabilities and reasonably assess the quality of a software project. Furthermore, there exist several easy-to-use (plug-and-play) free SAST tools (FSTs) to detect common classes of vulnerabilities and insecure practices. One of the well-known FST is CodeQL [32]. A recent study [64] shows that just running CodeQL

(with its default configuration) found more than 300 security vulnerabilities in open-source embedded software repositories. Also, the authors identified that only 4% use any sort of SAST in their repository. Our preliminary analysis also found that only 10% of critical OSS projects use CodeQL.

Many prior studies [14,26,35,54] focus mainly on software security vulnerabilities and try to investigate their effectiveness. Furthermore, most of these studies were performed on a small scale, raising concerns regarding the generalizability of their observations. OSS can contain other classes (e.g., improper configuration) of defects. *However, no existing work tries to understand the effectiveness of all categories of SAST tools at a large scale.* We argue that such a work will serve as guidance for tool developers and can potentially expose problems in the applicability of SAST tools on OSS.

3 Background

In this section, we provide the necessary background related to our methodology.

3.1 OpenSSF and Criticality Score

OpenSSF Open Source Security Foundation (OpenSSF) [3] is a community of software developers, security engineers, and more who are working together to secure open-source software. AlphaOmega [1] is an associated project of the OpenSSF, funded by Microsoft, Google, and Amazon, with a mission to protect society by catalyzing sustainable security improvements in the most critical open-source software projects and ecosystems.

Measuring Project Importance as Criticality Score OpenSSF created a mechanism to compute a criticality score [15] for GitHub repositories. Security analysts use this score to triage the security vulnerabilities by scanning large datasets. We use the *criticality score* to measure the importance of an open-source project. A project's criticality score is a number between 0 and 1. It is computed based on attributes, including its popularity, dependents, and level of activity. Ranges correspond to qualitative labels: 0.0-0.2 is considered low criticality, 0.2-0.4 is medium, 0.4-0.6 is high, 0.6-0.9 is critical, and above 0.9 is extremely critical. The Swift language frontend (with 2.4K stars) [6] has criticality scores of 0.51, indicating a high severity project. The Linux kernel (with 157K stars) [68] has a criticality score of 0.88, indicating a critical project. The Node.js runtime (with 97.6K stars) [5] has a score of 0.99, indicating an extremely critical project.

3.2 Static Application Security Testing (SAST)

SAST represents a class of techniques to find security issues in a given software. These techniques are *static*, i.e., they do not execute the target software. On

the contrary, dynamic techniques (e.g., random testing) execute the target and need an appropriate execution environment. A well-known category of SAST tools is code scanning tools, e.g., CodeQL, which find security vulnerabilities (e.g., buffer overflow) in source code. As these vulnerabilities are often severe, most of the SAST research focused on the code scanning tools. Consequently, the security community uses SAST synonymous with code scanning. However, there are also other classes of SAST tools (e.g., identifying misconfigurations), which also try to find import security issues but are not well-studied. For our study, we classify SAST tools into the following categories as shown in Table 3.

- **Source Code Scanning (SCS):** These are classic code scanning tools; they use pattern-matching (e.g.,) or flow-based static program analysis techniques (e.g., CodeQL) to find security vulnerabilities in programs' source code. For instance, CodeQL finds use-after-free vulnerabilities by performing a flow-sensitive analysis [33].
- **Mis-configurations (MC):** These tools focus on detecting sub-optimal or insecure configurations (e.g., hardcoding a private key) at both the source level and the project level. These are similar to SCS tools but also focus on non-code entities. Similarly, Binwalk [37] can be used for the detection of private hard-coded keys in the projects.
- **Quality and Best Practices (QS):** As the name suggests, this category of tools checks for quality issues at both the source code and project levels. For instance, Lizard [74] detects code with high complexity, a well-known proxy for potential bugs.
- **Software Statistics (SS):** These tools report interesting statistics that can serve as a proxy for anomalies or potential vulnerabilities. For instance, the SCC [19] tool computes the number of people required to maintain a given repository, i.e., "Estimated People Required", based on various software-driven metrics. The increase in this number indicates potential anomaly and refactoring opportunities.

These SAST tools often report the severity level of alerts to indicate the potential risk added to the codebase; for example, CodeQL alerts classify issues as "Error," "Warning," or "Note."

4 Study Design and Research Questions

Based on our motivation and background stated before, we have devised a study design as shown in Fig. 1. The execution of GitHub open source repositories is done on the Omega Analyzer. Based on the functionality of each static analysis tool of Omega Analyzer, its categorization is done. Based on running those static analysis tools on open source repositories, we have divided our study into three research questions as shown below. Research Question 1 (RQ1) focuses on the deployment of the Omega Analyzer to evaluate an extensive dataset of open source repositories. Research Question 2 (RQ2) is centered on a quantitative evaluation to interpret the errors and warnings distribution that

occur across different repositories while analyzing through Omega Analyzer. Research Question 3 (RQ3) explores the accuracy of SCS tools by analyzing their false positive rates in our repository dataset.

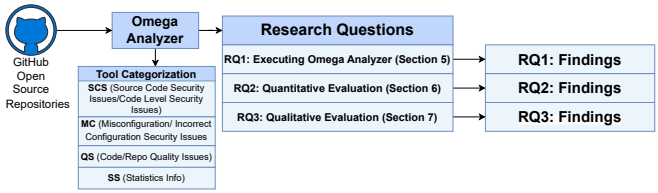

Fig. 1. Research Methodology

5 RQ1—Executing Omega Analyzer

Research Question 1 (RQ1) explores the deployment of Omega Analyzer on the dataset of open source software repositories, particularly focusing on the major programming languages used, the categorization of repositories and their primary languages, and the classification and language support of the Omega Analyzer underlying tools.

5.1 RQ1: Methods and Results

We developed an automated workflow for running Omega Analyzer on repositories. It takes the input of the GitHub repository for running the latest Omega Analyzer docker image on it and stores the results for review in SARIF format. We established self-hosted runners on eight machines, each utilizing an AWS EC2 m7a.2xlarge instance with 8 CPU cores, 32GB of memory, and 100GB of disk space, operating on Ubuntu 22.04. The workflow timeout was configured to three hours. The execution of OmegaAnalyzer across all repositories in our dataset took approximately eight days. We succeeded to run Omega Analyzer on 4,865 repositories. Failure factors include workflow timeouts (60), out-of-memory (2), and others (20), e.g. insufficient disk space. The details of our analysis are presented in the following sub-sections.

Open Source Software Dataset For this study, we analyzed a dataset of 4,947 high-criticality projects from the Open Source Security Foundation (OSSF). These projects were chosen for their significant impact on the open-source ecosystem. They span diverse applications and sectors, including infrastructure, web development, security, and data analysis.

- **Major Programming Languages and SLOC of repositories:** Table 1 lists the major programming languages used in these repositories, along with their Source Lines of Code (SLOC) statistics. It shows the number of repositories for each language and the maximum, mean, median, and minimum SLOC values. Python is the most used language with 383 repositories, followed by JSON (369) and JavaScript (322), highlighting their roles in data interchange and web development. JSON leads in SLOC with a maximum of 30 million lines, closely followed by Java and Go, reflecting their use in large-scale projects.
- **Repository Categories and Their Top Languages:** Table 2 shows the distribution of repositories across categories and the predominant programming languages used. Python and JavaScript are most frequent, especially in hacktoberfest, python, and react categories. C is prevalent in kubernetes, android, and linux, highlighting its system-level programming relevance. Go's presence in kubernetes and docker categories indicates its growing importance in cloud technologies. This data illustrates the diverse application of programming languages across domains.

Table 1. Major programming languages and SLOC of repositories.

Lang. ID	Language / Encoding Format	Num of Repo	SLOC			
			Max	Mean	Med	Min
L1	Python	383	6M	100K	40K	40
L2	JSON	369	30M	300K	60K	200
L3	JavaScript	322	3M	100K	60K	200
L4	Java	317	9M	400K	200K	2K
L5	PHP	284	2M	100K	40K	100
L6	Go	258	10M	500K	100K	2K
L7	C	250	20M	1M	100K	7K
L8	C++	228	4M	400K	200K	3K
L9	TypeScript	216	4M	200K	60K	700
L10	Markdown	166	6M	200K	20K	4
L11	Ruby	161	5M	100K	20K	1K
L12	C#	124	10M	400K	100K	5K
L13	POFile	109	9M	700K	200K	8K
L14	XML	88	10M	700K	200K	1K
L15	YAML	83	9M	300K	70K	40
L16	C/C++Header	76	5M	400K	100K	3K
L17	Rust	60	1M	100K	50K	3K
L18	Text	60	20M	1M	200K	3K
L19	HTML	57	5M	500K	100K	1K
L20	Scala	42	900K	100K	50K	6K
L21	QtLinguist	33	6M	800K	400K	10K
L22	Kotlin	30	2M	100K	60K	6K
L23	Haskell	26	600K	70K	20K	2K
L24	diff	25	4M	300K	200K	8K
L25	BourneShell	24	70K	20K	7K	100
L26	Swift	22	500K	100K	70K	7K
L27	SVG	19	2M	300K	100K	900
L28	CSV	19	3M	500K	200K	20K
L29	CSS	17	1M	200K	40K	3K
L30	Objective-C	12	300K	70K	20K	2K
–	Others	1067				

Results of Omega Analyzer Tools under each category Each tool within the Omega Analyzer plays a specific role, from static code analysis to detecting vulnerabilities, ensuring code quality, and safeguarding against security threats.

Table 2. Repository Categories and Their Top Languages

Category	# Repos	Top Language
hacktoberfest	807	Python
python	371	Python
javascript	264	JavaScript
java	211	Java
php	181	PHP
kubernetes	113	C
ruby	97	Ruby
react	93	JavaScript
c	91	C
go	80	Go
c-plus-plus	68	C++
rust	59	Rust
android	51	C
typescript	50	TypeScript
dotnet	43	C
linux	39	C
security	35	C
machine learning	30	Python
nodejs	29	JavaScript
docker	28	Go

– Omega Analyzer **Tools Categorization:** In our study, we have categorized the tools integrated into the Omega Analyzer based on the specific types of security and code quality issues they address. These categories are defined in Section 3. According to those categories we have assigned an ID number to each tool in Omega Analyzer. The ID number of each tool along with its description is shown in Table 3. Some interesting results of running these tools on repositories are shown below.

- **Source Code Scanning (SCS)**
  The DevSkim tool found an error in the Platform Helpers repository [56] involving a weak hash algorithm. This vulnerability compromises data integrity and security, making the system susceptible to attacks. The error is shown in Listing 1.1, where the line `checksum = sha1(hashlib_encode_data(__version__))` directly uses the version string for checksum calculation. If an attacker knows the version, they can manipulate or predict the checksum.

```
1 import re
2 from hashlib import sha1
3
4 def compute_project_checksum(config):
5     # rebuild when PIO Core version changes
6     checksum = sha1(hashlib_encode_data(__version__))
```

Listing 1.1. Hardcoding Version Information

- **Mis-configuration (MC)** The Semgrep tool detected common misconfiguration issues in the Microsoft Terminal Workflow repository [50]. Specifically, the line `uses: craigloewen-msft/GitGudSimilarIssues@main` in Listing 1.2 sources an action from a third-party repository without pinning it to a full-length commit SHA. Pinning to a full-length commit SHA is crucial as it ensures the action remains immutable, mitigating the risk of a backdoor being added to the action's repository.

```
1 steps:
2       - id: getBody
3         uses: craigloewen-msft/GitGudSimilarIssues@main
4         with:
5           issueTitle: ${{ github.event.issue.title }}
6           issueBody: ${{ github.event.issue.body }}
```

Table 3. Overview of Tools Integrated in Omega Analyzer and Their Categorization.

ID	Tool	Description
SCS Errors		
T1	DevSkim [47]	Identifies and fixes security issues in source code.
T2	NodeJsScan [9]	SAST tool for Node.js applications.
T3	CppCheck [45]	Static analysis for C and C++ code.
T4	CodeQL [32]	Automated code review and security analysis.
T5	SecretScanner [23]	Scans for secrets in code and file systems.
T6	Detect-Secrets [73]	Prevents secrets in code.
T7	Brakeman [21]	Security vulnerabilities detection in Ruby on Rails.
T8	Graudit [71]	Scans source code for security flaws.
T9	ILSpy [38]	.NET assembly browser and decompiler.
T10	npm audit [52]	Reviews npm projects for vulnerabilities in dependencies.
T11	Snyk Code [66]	Finds and fixes vulnerabilities in open-source dependencies.
T12	Bandit [59]	Finds common security issues in Python code.
T13	Semgrep [60]	Fast tool for bug detection and code standard enforcement.
MC Errors		
T14	ClamAV [67]	Antivirus engine for detecting malware.
T15	Yara [13]	Helps in malware identification and classification.
T16	Manalyze [41]	Static analyzer for PE files, malware detection.
MC Warnings		
T17	strace [43]	Monitors interactions between processes and the Linux kernel.
T18	OSS Gadget [49]	Tools for open-source intelligence.
T19	binwalk [37]	Searches binary images for embedded files and hidden data.
T20	ShhGit [58]	Scans GitHub for sensitive information.
T21	TBV [42]	Package verification for npm.
T22	Radare2 [12]	Binary analysis and reverse-engineering tool.
SS Statistics Info		
T23	Application Inspector [48]	Identifies and reports software features.
T24	SCC [19]	Code counter with complexity calculations.
QS Warnings		
T25	Lizard [74]	Analyzes code complexity and generates metrics.

```
7           repo: ${{ github.repository }}
8           similaritytolerance: "0.8"
```

Listing 1.2. GitHub Actions workflow step.

Another error is detected under this category in Yoast SEO WordPress plugin's components directory [75]. The Yoast SEO WordPress plugin's components directory, analyzed by Binwalk, contains exposed private keys and certificates. The RSA private key is exposed in the repository. See Listing 1.3. This oversight could compromise the security of the application by allowing unauthorized access or decryption of sensitive data.

```
1 -----BEGIN RSA PRIVATE KEY-----
2 MIIEpAIBAAKCAQEAn+5QIkyjSaeAt8o+htOoVaa9/rxU95ROYbpezlofm...
3 -----END RSA PRIVATE KEY-----
```

Listing 1.3. RSA Private Key exposed in repository.

- **Quality and Best Practices (QS):** The issue identified in this category is in the Facebook Hermes [27] repository relates to high code complexity, as detected by the tool Lizard. High code complexity can lead to difficulties in understanding, maintaining, and modifying the code, potentially increasing the risk of errors and reducing efficiency in development processes.
- **Software Statistics (SS)** The error reported in the SageMath repository [63] by the SCC tool falls under Statistics Info (SS). SCC estimated

that maintaining SageMath would require about 120 people, reflecting the project's complexity and scale. Key metrics include an estimated development cost of $111,404,644, a scheduled effort of 82.46 months, and 120 people required. These insights are vital for planning and resource allocation in software development.

– **Supported languages for Tools of** Omega Analyzer**:** The table 4 maps the applicability of Omega Analyzer tools (T1, T2, T3) across various programming languages (L1 to L30). The supported languages for the tools of OmegaAnalyzer illustrate the breadth of compatibility across different programming environments. For example, tools such as T1 and T3 support a wide range of languages, including Python (L1) and JavaScript (L3), which are commonly used in various applications. In contrast, other tools like T14 and T18 are compatible with fewer languages.

Table 4. Supported languages for Tools of Omega Analyzer

	T1	T2	T3	T4	T5	T6	T12	T13	T14	T17	T18	T19	T20	T23	T24	T25
L1	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓		✓
L2	✓	✓		✓			✓			✓					✓	
L3	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L4	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L5	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L6	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L7	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L8	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L9	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L10	✓			✓	✓	✓	✓			✓		✓			✓	
L11	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L12	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L17	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L19	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L20	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L22	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L23	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓			✓	
L26	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	
L29	✓	✓		✓	✓	✓	✓	✓		✓		✓			✓	
L30	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓			✓	

5.2 RQ1: Findings

– **Finding 1:** The execution of Omega Analyzer across 4,947 repositories revealed 60 workflow timeouts, 2 out-of-memory errors, and 20 disk space failures, resulting in a 98.3% success rate out of 4,947 repositories, indicating effective processing despite some resource constraints.

– **Finding 2:** In our dataset, the major programming language is Python with 383 repositories (Table 1), followed by JSON and JavaScript with 369 and 322 repositories respectively; JSON leads in Source Lines of Code (SLOC) with 30 million lines, followed by Java and Go, indicating their use in large-scale applications.

– **Finding 3:** The distribution of repositories across various categories, as detailed in Table 2 , indicates a dominant use of Python and JavaScript, especially in hacktoberfest, python, and react categories. Conversely, C is

prevalent in system-oriented categories such as kubernetes, android, and linux, while Go's significant presence in kubernetes and docker categories shows its increasing importance in cloud technologies. Overall, the data illustrates the diverse application of programming languages across different domains.

– **Finding 4:** Table 4 indicates the versatility of certain tools and suggests the need for specific tools tailored to handle the unique requirements of different programming languages, considering their varied usage and complexity levels.

6 RQ2—Quantitative Evaluation

Research Question 2 (RQ2) is centered on a quantitative evaluation to interpret the distribution of errors and warnings that occur across different repositories while being analyzed through the Omega Analyzer, focusing on the relationship between error distribution and criticality score, comparative error distribution across repositories by the Omega Analyzer tool, and warning distribution across repositories by analytical tools.

6.1 RQ2: Methods and Results

Following is a detailed analysis of our quantitative analysis.

Errors Distribution in relation to Criticality Score Figure 2 shows the distribution of errors in relation to the 'Criticality Score' of repositories. The x-axis represents the criticality score from 0-0.1 to 0.9-1.0, with higher scores indicating greater importance. The y-axis represents the percentage, ranging from 0% to 80%. Blue bars indicate the percentage of repositories within each score range, while red bars show the percentage of errors. Most repositories have a criticality score of 0.5-0.6, but the highest error percentage is in the 0.4-0.5 range. This suggests that certain criticality scores are more prone to errors regardless of the number of repositories.

Distribution of Alerts in Repositories by Omega Analyzer Figure 3 presents a comprehensive visualization of the distribution of alerts generated by the Omega Analyzer across a series of repositories. The x-axis represents the cumulative percentage of analyzed repositories, offering a progressive insight into the coverage of the dataset. The y-axis quantifies the number of alerts, distinguishing between errors (blue line) and warnings (red line). Notably, the graph serves as a diagnostic tool, highlighting the Omega Analyzer's capability in identifying and categorizing potential issues, thereby aiding in the prioritization of repository maintenance and code quality assurance.

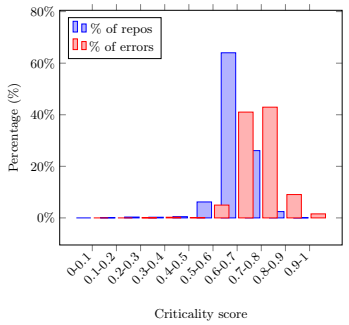

Fig. 2. Errors Distribution in relation to Criticality Score

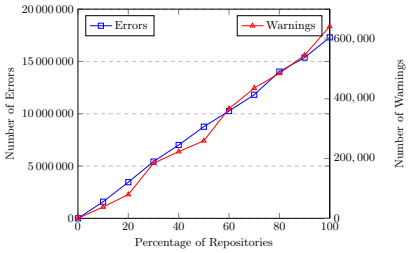

Fig. 3. Distribution of Alerts in Repositories by Omega Analyzer

Comparative Error Distribution Across Repositories by Omega Analyzer Figure 4 shows the cumulative percentage of errors detected by different analytical tools across repositories. The x-axis represents the percentage of repositories analyzed, and the y-axis indicates the percentage of errors detected. The varied trajectories highlight each tool's unique error detection patterns and sensitivities. The graph is essential for understanding error distribution and magnitude across repositories, with the y-axis reaching up to 15 million errors.

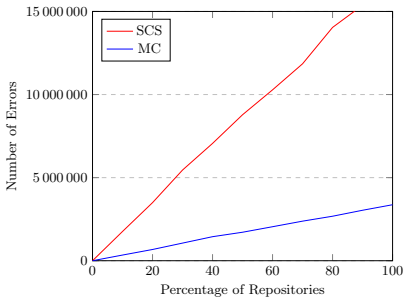

Fig. 4. Comparative Error Distribution Across Repositories by Omega Analyzer

Warning Distribution Across Repositories by Analytical Tools Figure 5 illustrates the distribution of warnings across repositories as analyzed by three tools: QS, SCS, and MC. The SCS tool, focusing on source code security issues, shows a sharp increase in warnings past the 50% mark of analyzed repositories. The MC tool, identifying misconfigurations, exhibits consistent but moderate

growth in warnings, suggesting widespread but stable configuration issues. The QS tool, targeting code and repository quality, displays a gradual increase in warnings, indicating a baseline level of quality issues. These trends reveal the diverse challenges in maintaining code quality and security across repositories.

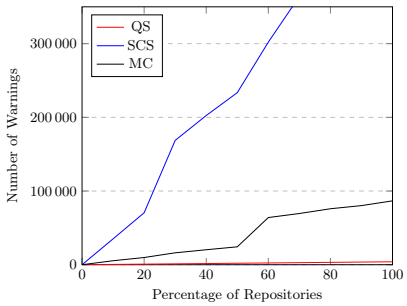

Fig. 5. Warning Distribution Across Repositories by Analytical Tools

6.2 RQ2: Findings

- **Finding 1:** The distribution of errors in relation to the criticality score of repositories shows that repositories with a criticality score of 0.4-0.5 have the highest percentage of errors, despite the majority of repositories having a criticality score in the 0.5-0.6 range (Figure 2).
- **Finding 2:** The Omega Analyzer identifies a significant number of errors and warnings across repositories, with a sharp increase in errors detected after analyzing 50% of the repositories. This suggests that errors are more prevalent in the latter half of the analyzed repositories.
- **Finding 3:** Different analytical tools, such as SCS and MC, exhibit unique error detection patterns, with SCS detecting a higher number of errors compared to MC. This difference reflects the distinct focuses and strengths of each tool, rather than a direct comparison of their effectiveness, as they are designed to address different aspects of code analysis (Figure 4).
- **Finding 4:** The distribution of warnings across repositories by analytical tools reveals that the SCS tool identifies a significantly higher number of warnings, especially after the 50% mark of analyzed repositories. This suggests that security-related issues become more pronounced in the latter half of the repository analysis (Figure 5).
- **Finding 5:** The QS tool, focused on detecting code and repository quality issues, shows a gradual increase in warnings, indicating a baseline level of code quality is maintained across repositories, but still highlighting prevalent quality issues (Figure 5).
- **Finding 6:** The moderate but consistent growth in warnings identified by the MC tool across analyzed repositories implies that configuration errors are

widespread but do not vary dramatically, emphasizing the need for consistent configuration management (Figure 5).

7 RQ3—Qualitative Evaluation

In this section, we assess the quality of SCS tools. Specifically, we want to study the false positive rates of these tools on our dataset of repositories. SCS tools are pivotal in modern software development, providing automated means to detect potential vulnerabilities early in the development lifecycle. However, the efficacy of these tools is often diminished by false positives – instances where a tool reports a security issue that is not actually present in the codebase. High false positive rates can lead to wasted time and resources as developers must spend considerable time verifying and dismissing false alarms, which hinders their productivity and delays the development process.

In the rest of this section, we will first present our methodology, followed by the results.

7.1 RQ3: Methods and Results

Methods Categorizing alerts into true and false positives requires significant manual effort. Given the large number of repositories and alerts, it is infeasible to analyze them all. To address this, we implement a random sampling approach to maintain manageability while ensuring a representative evaluation. For each SCS tool, we randomly sample 10 repositories that each has fewer than 20 errors or warnings reported.

Each reported issue is manually reviewed to determine its accuracy. This involves examining the code in question to ascertain whether the reported issue is a true positive (a genuine security vulnerability) or a false positive (an incorrectly identified issue).

Results We managed to obtain the false positive rate for five SCS tools. Unfortunately, we did not have enough data to evaluate other tools due to the following reasons: 1) No results in the dataset: Some tools did not yield any results in any of the repositories within our dataset; 2) Lack of source line Information: Some tools failed to report the source line number for the issues detected, making it challenging for us to verify and categorize the results accurately.

The false positive rates for the evaluated tools are summarized in Table 5. These results indicate significant variability in the accuracy of the evaluated SCS tools. Tool T4 demonstrated the highest accuracy with a false positive rate of only 9%, making it the most reliable among the tools tested. In contrast, Tools T1 and T2 had the highest false positive rates, each exceeding 60%, which suggests a need for improvement in their detection algorithms.

To illustrate the nature of false positives encountered during our evaluation, Listing 1.4 shows an example of a false positive reported by T1. The tool flagged a piece of code as a potential security vulnerability, but upon manual review, it

Table 5. Number of true positives, false positives, and false positive rate for the five evaluated SCS tools.

	#TP	#FP	FP%
T1	28	50	64%
T2	9	17	65%
T4	51	5	9%
T12	53	28	35%
T13	45	35	44%

was determined to be a false positive. Specifically, this tool incorrectly identified the use of the SHA-512 hash algorithm as weak or broken.

Overall, our findings highlight the importance of evaluating and selecting SCS tools carefully, as the effectiveness of these tools can vary greatly. Accurate tools can significantly aid in identifying genuine security vulnerabilities, while those with high false positive rates can burden developers with unnecessary reviews and potentially lead to overlooked issues.

```
1 "node_modules/normalize-package-data": {
2   "version": "5.0.0",
3   "resolved": "https://registry.npmjs.org/normalize-package-data/-/normalize-package-data-5.0.0.tgz",
4   "integrity": "sha512-h9iPVIfrVZ9wVYQnxFgtw1ugSvGEMOlyPWWtm8BMJhnwyEL/FLbYbTY3V3PpjI/
        BUK67n9PEWDu6eHzu1fB15Q==",
5   ...
6 },
```

Listing 1.4. An example of a false positive reported by T1. T1 flagged the hash algorithm used for integrity checking of the package as weak or broken. However, the integrity field uses SHA-512, which is considered secure.

7.2 RQ3: Findings

- **Finding 1:** There is significant variability in the accuracy of the evaluated tools. Tool T4 had the lowest false positive rate at 9%, indicating high reliability, while Tools T1 and T2 had the highest false positive rates, each exceeding 60%.
- **Finding 2:** High false positive rates can mislead developers. For example, one tool incorrectly flagged the secure SHA-512 hash algorithm as weak, illustrating the need for refinement in detection algorithms to reduce unnecessary reviews and potential oversight of genuine issues.

8 Related Work

This literature review examines three main areas: firstly, it looks into studies that explore the security vulnerabilities in open-source repositories, identifying the key risks and challenges involved. Secondly, it evaluates the effectiveness of existing Static Application Security Testing (SAST) tools. Finally, it reviews efforts that apply these SAST tools to open-source software.

8.1 Security Vulnerabilities in Open Source Repositories

Recent studies have significantly advanced our understanding of security vulnerabilities in open-source software repositories. Research has included case studies on major projects like Apache HTTP Server and Apache Tomcat, revealing specific security fixes and preventive measures [55]. The reliability of data in vulnerability repositories has been critically evaluated, highlighting inconsistencies and gaps in current databases [40]. Tools like CVEfixes and VCCFinder have been developed to automate the collection of vulnerabilities and assist in code audits by mining software repositories [17] [76]. Empirical analyses have shed light on the nature and frequency of security issues reported in open-source projects, with some focusing on mining threat intelligence from issues and bug reports [76]. Efforts to generate datasets from vulnerable source code have enhanced the resources available for understanding and mitigating these risks [61] [34] [57]. Studies have also proposed models to estimate and predict security risks associated with open-source packages, contributing to more informed decision-making in software development [65] [72]. Moreover, assessments of the impact of vulnerabilities in software libraries have been complemented by investigations into the release practices and secret integration channels of open-source packages [39] [62].

8.2 Evaluating the Effectiveness of SAST Tools

Research has extensively evaluated the effectiveness of Static Application Security Testing (SAST) tools in diverse programming environments. Key studies have benchmarked SAST tools for C, revealing strengths and limitations [30] [29]. Innovative uses, such as employing AI models like ChatGPT for static security testing, have been examined for their practicality [16]. Empirical work has shed light on the performance and reliability of these tools, particularly through security warnings [11]. Additionally, comparative analyses have highlighted SAST tools' capabilities in Java and distributed applications, contrasting them with dynamic methods to evaluate thoroughness [44] [24]. The integration of SAST with reverse engineering for binary executables also illustrates a comprehensive approach to security [25].

8.3 Application of SAST Tools in Open-Source Software Environments

Mingjie Shen et al. [64] studied 258 popular EMBOSS projects using GitHub's CodeQL, finding 540 defects, 74% of which were probable security vulnerabilities, highlighting SAST tools' effectiveness and low false positive rate (23%). Feras Al Kassar et al. [10] examined the impact of code patterns on security testing in web applications, identifying over 270 patterns that hinder static analysis and discovering 440 new vulnerabilities across 48 projects. A. Nguyen-Duc et al. [51] found that combining SAST tools enhances performance in an open-source e-government project. MM Casanova Páez et al. [20] reviewed various application

security testing tools, emphasizing automation, ease of use, and accuracy, and found no direct correlation between commercial tools and higher effectiveness.

Additionally, C. Gentsch et al. [29] evaluated multiple open-source Static Analysis Security Testing (SAST) tools for C, including AdLint, Clang-Tidy, and others. Using a methodology involving file-by-file analysis and a comprehensive database to track outputs, the study confirmed the tools' efficacy by comparing findings against the Juliet Test Suite. This rigorous approach assessed overall tool accuracy, overlap, and usability in various real-world and synthetic environments. Further, F. Mateo Tudela and J.R. Bermejo Higuera [46] analyzed the effectiveness of combining SAST, DAST, and IAST tools against the OWASP Top Ten vulnerabilities. Using combinations like Fortify+Arachni+CCE, they achieved notable success across various security levels in web applications.

9 Discussion and Future Work

We will responsibly disclose all critical vulnerabilities found, such as exposed private keys, to the respective maintainers. In Section 7, we limited our analysis to repositories with fewer than 20 errors or warnings, which may bias results by excluding projects with more issues. Future work will expand the scope to include repositories with a broader range of issues and develop a semi-automated system to classify false positives, enhancing the scalability and accuracy of our analysis.

10 Conclusion

In conclusion, this paper provides a detailed examination of the quality and security of open-source software (OSS) using 24 open-source static analysis tools through Omega Analyzer. By analyzing 4,865 significant OSS projects, we have uncovered numerous insights that underscore the effectiveness of these tools in identifying security vulnerabilities and enhancing code quality. The findings from our comprehensive study not only highlight the pivotal role of static analysis tools in securing OSS repositories but also suggest future research directions aimed at refining these tools and strategies.

Acknowledgements

This research was partly supported by the National Science Foundation (NSF) under Grant CNS-2340548. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

1. Alpha Omega – Linux Foundation Project, `https://alpha-omega.dev/`

2. CodeQL, https://codeql.github.com/
3. Open Source Security Foundation – Linux Foundation Projects, https://openssf.org/
4. Understanding GitHub Actions, https://docs.github.com/_next/data/0DKyBPMqZhPYD1Lsg3qKt/en/free-pro-team@latest/actions/learn-github-actions/understanding-github-actions.json?versionId=free-pro-team%40latest&productId=actions&restPage=learn-github-actions&restPage=understanding-github-actions
5. Node.js. https://github.com/nodejs/node (Sep 2023), original-date: 2014-11-26T19:57:11Z
6. SwiftSyntax. https://github.com/apple/swift-syntax (Sep 2023), original-date: 2018-07-31T23:19:58Z
7. Fact Sheet: Biden-Harris Administration Releases End of Year Report on Open-Source Software Security Initiative | ONCD (Jan 2024), https://www.whitehouse.gov/oncd/briefing-room/2024/01/30/fact-sheet-biden-harris-administration-releases-end-of-year-report-on-open-source-software-security-initiative/
8. nodejs/node (Jun 2024), https://github.com/nodejs/node, original-date: 2014-11-26T19:57:11Z
9. Abraham, A.: Nodejsscan (2023), https://github.com/ajinabraham/NodeJsScan, accessed: 2024-05-18
10. Al Kassar, F., Clerici, G., Compagna, L., Balzarotti, D., Yamaguchi, F.: Testability tarpits: the impact of code patterns on the security testing of web applications. In: NDSS Symposium 2022. Internet Society, San Diego, California, USA (2022)
11. Aloraini, B., Nagappan, M., German, D.M., Hayashi, S., Higo, Y.: An empirical study of security warnings from static application security testing tools. Journal of Systems and Software **158**, 110427 (2019)
12. Alvarez, S.: Radare2 (2006), https://www.radare.org/, accessed: 2024-05-18
13. Alvarez, V.: Yara (2024), https://virustotal.github.io/yara/, accessed: 2024-05-18
14. Arusoaie, A., Ciobâca, S., Craciun, V., Gavrilut, D., Lucanu, D.: A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In: 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 161–168 (2017). https://doi.org/10.1109/SYNASC.2017.00035
15. Arya, A., Brown, C., Pike, R., The Open Source Security Foundation: Open Source Project Criticality Score. https://github.com/ossf/criticality_score (Mar 2023), original-date: 2020-11-17T16:14:23Z
16. Bakhshandeh, A., Keramatfar, A., Norouzi, A., Chekidehkhoun, M.M.: Using chatgpt as a static application security testing tool. arXiv preprint arXiv:2308.14434 **N/A**, N/A (2023)
17. Bhandari, G., Naseer, A., Moonen, L.: Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 30–39. Association for Computing Machinery, Athens, Greece (2021)
18. Bonaccorsi, A., Rossi, C.: Why open source software can succeed. Research policy **32**(7), 1243–1258 (2003)
19. Boyter, B.: Scc (2018), https://github.com/boyter/scc, accessed: 2024-05-18
20. Casanova Páez, M.M.: Application security testing tools study and proposal. N/A **N/A**, N/A (2021)

21. Collins, J.: Brakeman (2010), https://brakemanscanner.org/, accessed: 2024-05-18
22. Cybersecurity and Infrastructure Security Agency (CISA): Government and industry partners publish fact sheet for organizations using open source software (2023), https://www.cisa.gov/news-events/news/government-and-industry-partners-publish-fact-sheet-organizations-using-open-source-software, accessed: 2024-06-13
23. Deepfence: Secretscanner (2020), https://github.com/deepfence/SecretScanner, accessed: 2024-05-18
24. Dencheva, L.: Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools. Ph.D. thesis, Dublin, National College of Ireland (2022)
25. Devine, T.R., Campbell, M., Anderson, M., Dzielski, D.: Srep+ sast: A comparison of tools for reverse engineering machine code to detect cybersecurity vulnerabilities in binary executables. In: 2022 International Conference on Computational Science and Computational Intelligence (CSCI). pp. 862–869. IEEE, Las Vegas, NV, USA (2022)
26. Esposito, M., Falaschi, V., Falessi, D.: An extensive comparison of static application security testing tools (2024)
27. Facebook: Hermes javascript engine. https://github.com/facebook/hermes (2024), accessed: 2024-04-30
28. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Security testing: A survey. In: Advances in Computers, vol. 101, pp. 1–51. Elsevier (2016)
29. Gentsch, C.: Evaluation of open source static analysis security testing (sast) tools for c. N/A **N/A**, N/A (2020)
30. Gentsch, C., Krishnamurthy, R., Heinze, T.S.: Benchmarking open-source static analyzers for security testing for c. In: Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part IV 9. pp. 182–198. Springer, Rhodes, Greece (2021)
31. Ghazaly, N.M.: Learning the idea behind sast (static application security testing) and how it functions. International Journal of Management and Engineering Research **1**(1), 01–04 (2021)
32. GitHub: Codeql (2019), https://securitylab.github.com/tools/codeql, accessed: 2024-05-18
33. GitHub: Potential use after free. https://codeql.github.com/codeql-query-help/cpp/cpp-use-after-free/ (2024), accessed: 2024-06-24
34. Gkortzis, A., Mitropoulos, D., Spinellis, D.: Vulinoss: a dataset of security vulnerabilities in open-source systems. In: Proceedings of the 15th International conference on mining software repositories. pp. 18–21. ACM, Gothenburg, Sweden (2018)
35. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Information and Software Technology **68**, 18–33 (2015)
36. Hauge, Ø., Ayala, C., Conradi, R.: Adoption of open source software in software-intensive organizations–a systematic literature review. Information and Software Technology **52**(11), 1133–1154 (2010)
37. Heffner, C.: binwalk (2010), https://github.com/ReFirmLabs/binwalk, accessed: 2024-05-18

38. ICSharpCode: Ilspy (2011), https://github.com/icsharpcode/ILSpy, accessed: 2024-05-18
39. Imtiaz, N., Khanom, A., Williams, L.: Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. IEEE Transactions on Software Engineering **49**(4), 1540–1560 (2022)
40. Jiang, Y., Jeusfeld, M., Ding, J.: Evaluating the data inconsistency of open-source vulnerability repositories. In: Proceedings of the 16th International Conference on Availability, Reliability and Security. pp. 1–10. ACM, Vienna, Austria (2021)
41. JusticeRage: Manalyze (2010), https://github.com/JusticeRage/Manalyze, accessed: 2024-05-18
42. Konves, S.: Tbv (2019), https://github.com/verifynpm/tbv, accessed: 2024-05-18
43. Levin, D.V.: strace (1992), https://strace.io/, accessed: 2024-05-18
44. Li, K., Chen, S., Fan, L., Feng, R., Liu, H., Liu, C., Liu, Y., Chen, Y.: Comparison and evaluation on static application security testing (sast) tools for java. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 921–933. ACM, San Francisco, CA, USA (2023)
45. Marjamäki, D.: Cppcheck (2007), http://cppcheck.sourceforge.net/, accessed: 2024-05-18
46. Mateo Tudela, F., Bermejo Higuera, J.R., Bermejo Higuera, J., Sicilia Montalvo, J.A., Argyros, M.I.: 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)
47. Microsoft: Devskim (2017), https://github.com/microsoft/DevSkim, accessed: 2024-05-18
48. Microsoft: Applicationinspector (2019), https://github.com/microsoft/ApplicationInspector, accessed: 2024-05-18
49. Microsoft: Ossgadget (2020), https://github.com/microsoft/OSSGadget, accessed: 2024-05-18
50. Microsoft: Workflow configuration for similar issues in microsoft terminal. https://github.com/microsoft/terminal/blob/main/.github/workflows/similarIssues.yml (2024), accessed: 2024-04-30
51. Nguyen-Duc, A., Do, M.V., Hong, Q.L., Khac, K.N., Quang, A.N.: On the adoption of static analysis for software security assessment–a case study of an open-source e-government project. computers & security **111**, 102470 (2021)
52. npm, I.: npm audit (2018), https://docs.npmjs.com/cli/v7/commands/npm-audit, accessed: 2024-05-27
53. Onarcan, M.O., Fu, Y., et al.: A case study on design patterns and software defects in open source software. Journal of Software Engineering and Applications **11**(05), 249 (2018)
54. Oyetoyan, T.D., Milosheska, B., Grini, M., Soares Cruzes, D.: Myths and facts about static application security testing tools: an action research at telenor digital. In: Agile Processes in Software Engineering and Extreme Programming: 19th International Conference, XP 2018, Porto, Portugal, May 21–25, 2018, Proceedings 19. pp. 86–103. Springer International Publishing (2018)
55. Piantadosi, V., Scalabrino, S., Oliveto, R.: Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 68–78. IEEE, Xi'an, China (2019)

56. PlatformIO: Project helpers for platformio. https://github.com/platformio/platformio-core/blob/develop/platformio/project/helpers.py (2024), accessed: 2024-04-30
57. Ponta, S.E., Plate, H., Sabetta, A., Bezzi, M., Dangremont, C.: A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). pp. 383–387. IEEE, Montreal, QC, Canada (2019)
58. Price, P.: Shhgit (2018), https://github.com/eth0izzle/shhgit, accessed: 2024-05-18
59. PyCQA: Bandit (2013), https://github.com/PyCQA/bandit, accessed: 2024-05-18
60. r2c: Semgrep (2020), https://semgrep.dev/, accessed: 2024-05-18
61. Raducu, R., Esteban, G., Rodriguez Lera, F.J., Fernández, C.: Collecting vulnerable source code from open-source repositories for dataset generation. Applied Sciences **10**(4), 1270 (2020)
62. Ramsauer, R., Bulwahn, L., Lohmann, D., Mauerer, W.: The sound of silence: Mining security vulnerabilities from secret integration channels in open-source projects. In: Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop. pp. 147–157. ACM, Virtual Event (2020)
63. SageMath: Sagemath mathematical software system. https://github.com/sagemath/sage (2024), accessed: 2024-04-30
64. Shen, M., Pillai, A., Yuan, B.A., Davis, J.C., Machiry, A.: An empirical study on the use of static analysis tools in open source embedded software. arXiv preprint arXiv:2310.00205 **N**/**A**(N/A), 1–14 (2023)
65. Smith, L.J.: Estimating Security Risk in Open Source Package Repositories: An Empirical Analysis and Predictive Model of Software Vulnerabilities. Ph.D. thesis, Capella University (2019)
66. Snyk: Snyk code (2020), https://snyk.io/product/snyk-code/, accessed: 2024-05-27
67. Talos, C.: Clamav (2024), https://www.clamav.net/, accessed: 2024-05-18
68. Torvalds, L.: torvalds/linux. https://github.com/torvalds/linux (Sep 2023), original-date: 2011-09-04T22:48:12Z
69. Torvalds, L.: torvalds/linux (Jun 2024), https://github.com/torvalds/linux, original-date: 2011-09-04T22:48:12Z
70. Ven, K., Verelst, J., Mannaert, H.: Should you adopt open source software? IEEE software **25**(3), 54–59 (2008)
71. wireghoul: Graudit (2010), https://github.com/wireghoul/graudit, accessed: 2024-05-18
72. Xu, R., Tang, Z., Ye, G., Wang, H., Ke, X., Fang, D., Wang, Z.: Detecting code vulnerabilities by learning from large-scale open source repositories. Journal of Information Security and Applications **69**, 103293 (2022)
73. Yelp: Detectsecrets (2017), https://github.com/Yelp/detect-secrets, accessed: 2024-05-18
74. Yin, T.: Lizard (2014), https://github.com/terryyin/lizard, accessed: 2024-05-18
75. Yoast: Components directory of the yoast seo wordpress plugin. https://github.com/Yoast/wordpress-seo/tree/trunk/apps/components (2024), accessed: 2024-04-30
76. Zahedi, M., Ali Babar, M., Treude, C.: An empirical study of security issues posted in open source projects. In: Proceedings of the 51st Hawaii International Conference on System Sciences (HICSS). pp. 5504–5513. IEEE, Hawaii, USA (2018)