Evaluating the Use of Open-Source and Standalone SAST Tools for Detecting Vulnerabilities in C/C++ Projects

Valdeclébio Farrapo, Emanuel Rodrigues,
José Maria Monteiro and Javam Machado
Computer Science Department, Federal University of Ceará, Brazil

Keywords: SAST Tools, Vulnerability Detection, C/C++ Code.

Abstract: Detecting security vulnerabilities in the source code of software systems is one of the most significant challenges in the field of information security. In this context, the Open Web Application Security Project (OWASP) defines Static Application Security Testing (SAST) tools as those capable of statically analyzing the source code, without executing it, to identify security vulnerabilities, bugs, and code smells during the coding phase, when it is relatively inexpensive to detect and resolve security issues. However, most well-known SAST tools are commercial and web-based, requiring the upload of the source code to a "trusted" remote server. In this paper, our goal is to investigate the viability of using open-source standalone SAST tools for detecting security vulnerabilities in C/C++ projects. To achieve our goal, we conduct an empirical study in which we examine 30 large and popular C/C++ projects using two different state-of-the-art open-source and standalone SAST tools. The results demonstrate the potential of using open-source standalone SAST tools as a means to evaluate the security risks of a software product without manually reviewing all the warnings.

1 INTRODUCTION

Nowadays, almost every daily task relies, to some extent, on software, ranging from electronic shopping to smart homes and autonomous vehicles. Naturally, the increase in the amount of source code implies greater security requirements in software development. In this context, a software security vulnerability can be defined as a coding flaw present in the source code of a software application that can be exploited by an attacker to gain unauthorized access, expose information, compromise data integrity, or disrupt and alter its functionality[1].

Security vulnerabilities have caused significant financial losses to companies and threatened critical security infrastructures (Parizi et al., 2020). The proliferation of cyber warfare further elevates the importance of preventing vulnerabilities. Therefore, businesses and governments require effective solutions to identify and fix vulnerabilities before valuable information is compromised (Dias et al., 2023).

Traditionally, software security is implemented during the testing phase, heavily relying on manual intervention, which delays its delivery and deployment. This practice has led organizations to neglect security, endangering both the software and customer data (Ahmed, 2019). Detecting software defects after a product's release forces companies to bear the costs of repairs and legal proceedings while also damaging their reputation. Consequently, one of the most critical tasks for organizations today is ensuring the security of their software products. Detecting vulnerabilities during the development phase, before the software is deployed in production environments, is crucial. Therefore, it is essential for software development teams to focus on identifying and resolving vulnerabilities in the source code. However, manual inspection of software is impractical, as the process is tedious and may not yield the expected results. To make vulnerability detection more efficient in terms of time, coverage, and number of vulnerabilities identified, automated methods have been proposed (Santos and de Santana Oliveira, 2017).

There are two widely used automated methods for detecting vulnerabilities in source code: (1) Static Code Analysis and (2) Dynamic Code Analysis. Both static and dynamic analyses audit the entire software

[1] https://cartilha.cert.br/

Farrapo, V., Rodrigues, E., Monteiro, J. M. and Machado, J.
Evaluating the Use of Open-Source and Standalone SAST Tools for Detecting Vulnerabilities in C/C++ Projects.
DOI: 10.5220/0013483500003929
In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 1*, pages 394-401
ISBN: 978-989-758-749-8; ISSN: 2184-4992

to identify vulnerabilities. Static code analysis scans the entire source code of a system to uncover potential security vulnerabilities. This method infers the behavior of a program without executing it. Static code analysis detects vulnerabilities while the software is still in the development phase. On the other hand, dynamic code analysis is performed after the developer executes the source code. As observed by many software developers and researchers, static code analysis has proven to be more efficient than dynamic code analysis in identifying software security vulnerabilities (Macedo and Salgado, 2015).

Static Application Security Testing (SAST) tools analyze a snippet of code or its compiled version to identify security issues, including suspicious constructs, unsafe API usage, dangerous runtime errors, bugs, duplications, and bad smells (Wheeler, 2015). SAST tools assist developers in detecting security vulnerabilities during the coding phase, where identifying and addressing security issues in the source code is relatively less expensive. For this reason, SAST tools are becoming increasingly crucial in the software development lifecycle (Fagan et al., 2020). Several tools claim to perform "security analysis", including AppScan Source, Checkmarx, Fortify, Semgrep, Bandit, and SonarQube. However, most SAST tools are commercial and web-based, requiring the source code to be sent to a "trusted" remote server.

In this paper, our goal is to investigate the feasibility of using open-source and standalone SAST tools for detecting security vulnerabilities in C/C++ projects. To achieve this goal, we conducted an empirical study in which we examined 30 C/C++ projects using two open-source and standalone SAST tools: *Flawfinder* (FF) [2] and *Visual Code Grepper* (VCG) [3]. Additionally, the study adopted guidelines provided by the *Common Weakness Enumeration (CWE)* [4] and the *Open Web Application Security Project (OWASP)* [5] to identify and analyze vulnerabilities in the source codes made available in these repositories. The results demonstrate the potential of using open-source and standalone SAST tools as a means of evaluating the security risks of a software product without manually reviewing all findings.

The remainder of this paper is organized as follows. Section 2 discusses the main related works. Section 3 presents the methodology used in this research. Section 4 analyzes the results obtained. Finally, Section 5 presents the conclusions and points to directions for future work.

[2]https://github.com/david-a-wheeler/flawfinder

[3]https://github.com/nccgroup/VCG

[4]https://cwe.mitre.org/

[5]https://owasp.org/

2 RELATED WORKS

In (Baca et al., 2008), the authors used a commercial SAST tool called *Coverity Prevent* to evaluate three private C++-based telecommunications software projects from the same company. The case study involved implementing a code security verification step within the software development lifecycle, where developers used the reports generated by the *Coverity Prevent* tool to fix vulnerabilities. The authors concluded that incorporating the security verification step and using the analyzed tool provided improvements in the quality and security of the developed software.

In the study presented in (Woody et al., 2020), the authors discuss the challenges of adopting *DevSecOps* and the importance of incorporating a security step into the software development lifecycle within organizations of the United States Department of Defense. They found that using *DevSecOps* allowed for mitigating security risks by detecting and analyzing vulnerabilities in an automated pipeline that accelerated the secure delivery process of the software.

In (de França and da Silva, 2022), the authors describe a case study aimed at comparing two development processes. The first process involves *DevOps* without the security analysis step, while the second corresponds to *DevSecOps*, which includes a dedicated security step. The results revealed that, in both pipelines, the product was delivered. However, in the process using *DevSecOps*, it was possible to avoid delivering a vulnerable application, demonstrating that integrating continuous security into *DevOps* workflows is not only feasible but can also provide significant benefits to organizations.

In the work presented in (Rahul et al., 2019), the authors integrated code security analysis as part of the software development process, preventing attacks and creating a secure environment and a more protected system. They used the open-source platform JENKINS and code analysis plugins that perform SAST checks.

In (Shi et al., 2024), a methodology was proposed to produce and optimize a knowledge graph that aggregates information from common threat databases (CVE, CWE, and CPE). The threat knowledge graph was applied to predict associations between threats, specifically between products and vulnerabilities. The authors demonstrated the ability of the threat knowledge graph to uncover many associations that are currently unknown.

In (Esposito et al., 2024), the authors aim to evaluate the effectiveness of vulnerability identification mechanisms based on SASTs versus machine learning alternatives. They investigated the use of eight

SAST tools (FindSecBugs, Jlint, Infer, PMD, SonarQube, Snyk Code, Spotbugs, VCG) on projects developed in Java. There were approximately 1.5 million test executions on a controlled codebase, establishing a reliable benchmark for evaluating the use of SAST tools. The results indicated that SAST tools detect a small variety of vulnerabilities. Contrary to common belief, SAST tools showed high precision but fell short in recall. The work suggests that improving recall along with increasing the number of vulnerabilities detected should be the main focus to enhance SAST tools.

Table 1 presents a comparative analysis between the current paper and the main related works. In the "Rep" column, we list the number of repositories (projects) used in each study. The "Language" column shows the programming languages used in each study. The "Tools" column lists the SAST tools used in the referred works. The "Open Source" column indicates whether the SAST tools used are open source or not. The "Standalone" column indicates whether the SAST tools used operate independently, i.e., without the need to upload the source code to a remote "trusted" server.

Unlike the work presented in (Baca et al., 2008), our research uses 30 repositories available on GitHub, involving projects with different contexts and sizes. In contrast to the study discussed in (Esposito et al., 2024), our work focused on open-source tools. The work presented in (Shi et al., 2024) discussed the creation of an association mechanism between CVE, CWE, and CPE, whereas in our research, we collected the vulnerabilities detected in the scans and cataloged them according to their corresponding CWEs, aiming to identify which CWEs the investigated SAST tools can detect. Finally, our work explored SAST tools that are executed locally (standalone), meaning they do not require uploading the source code to "trusted" remote servers. In practice, this need to upload the source code to third-party environments makes the use of these SAST tools unfeasible due to obvious concerns of trust and control. No company developing innovative technologies would allow its source code to be sent to third-party environments. The exclusive use of standalone SAST tools differentiates our work from all others.

3 METHODOLOGY

In this study, we adopted the action-research methodology (Baldissera, 2001), combining research with practical action. Below, we describe in detail all the steps of the methodology used.

Initially, the first challenge was to select the SAST tools to be used. To do this, we defined the following selection criteria:

1. **Free of Charge.** The tools should be free.
2. **Open Source.** The tools should be open-source.
3. **Standalone.** The tools should be capable of running locally, without the need to upload the source code to a remote server.
4. **Support for C/C++.** The tools should support analyzing code in C/C++.

After thorough research, we found two SAST tools that met all the selection criteria: *Flawfinder* (FF) and *Visual Code Grepper* (VCG). Both tools were integrated into our development environment.

The next step involved selecting the software projects to be evaluated. We chose to use public repositories available on *GitHub*, applying the following selection criteria:

1. **Availability.** The repositories should be public to allow access to the source code.
2. **Size.** The repositories should have varied sizes to ensure the research is comprehensive and representative.
3. **Programming Language.** The projects should have at least 70% of their code written in C/C++.

After this screening process, 30 repositories were selected. For each repository, we retrieved the code from 4 distinct commits (or versions): the first commit, the commits corresponding to the first and third quartiles in terms of project duration, and the last commit. This resulted in a total of 120 code snapshots. For each snapshot, we ran the *Flawfinder* (FF) and *Visual Code Grepper* (VCG) tools, configured to identify all possible vulnerabilities. This process generated a total of 240 security reports.

It is important to highlight that, in some repositories, the first commit was only used to create the repository, without submitting source code. In these cases, we discarded the first commit and replaced it with the second commit. Additionally, we noticed that some repositories were frequently updated, while others had not been updated for a long time. Therefore, we did not use time as a criterion for selecting commits, but rather the chronological order of the commits.

This methodology ensures a comprehensive and detailed analysis of code security quality in the selected repositories, using open-source, locally executed analysis tools, which facilitates practical application and reproducibility of the research.

Table 1: Comparative analysis of related works.

Work	Rep	Language	Tools	Open Source	Standalone
Baca et al. (2008)	3	C++	Coverity Prevent	NO	NO
Woody et al. (2020)	1	-	-	-	-
França e Silva (2022)	1	JS	Snyk CLI e Njsscan	YES	NO
Rahul et al. (2019)	1	-	Jenkins	NO	NO
Shi et al. (2024)	-	-	-	-	-
Esposito et al. (2024)	-	JAVA	FindSecBugs Jlint Infer PMD SonarQube Snyk Code Spotbugs VCG	NO	NO
This Article	30	C\C++	Flawfinder e Visual Code Grepper	YES	YES

After the selection of the commits, we began the testing process, which was executed in isolation, using one tool at a time. During the execution of the tests, the tools were configured to present the results in table format, and the data was stored for later analysis. It is important to note that each tool has its own characteristics, both in terms of execution mode, operation, and result presentation, as well as the levels of criticality and detection criteria for vulnerabilities. Below, we present the main differences between the tools used.

The SAST tool *Flawfinder* (FF) supports only the C/C++ language. It is quite simple and can be executed with a single command line. The results can be directly presented in table format, including the level of criticality and two corresponding CWEs for each detected vulnerability.

On the other hand, the SAST tool *Visual Code Grepper* (VCG) supports C/C++, Java, PL/SQL, C#, VB, PHP, and COBOL. Unlike *Flawfinder*, it was not possible to run *Visual Code Grepper* (VCG) via command line in a *Windows* environment, and it had to be executed from its graphical user interface. The tool is simple, lightweight, and user-friendly. The results can be reported directly in table format. Additionally, *Visual Code Grepper* (VCG) allows the generation of graphs based on the obtained results. The vulnerabilities are identified by criticality levels, but they are not cataloged by their corresponding CWEs. Therefore, after generating the reports with *VCG*, we manually analyzed each reported vulnerability and then associated two CWEs with each one. This strategy was used to allow comparison between the results from *Flawfinder* (FF) and *Visual Code Grepper* (VCG).

4 RESULTS

In this section, we present the results obtained from 240 tests conducted using the *Flawfinder* and *Visual Code Grepper* tools. The results are analyzed from three perspectives: the total number of vulnerabilities detected by each tool, the total number of unique detections (i.e., excluding duplicates), and the classification of vulnerabilities based on their corresponding CWEs.

Table 2 summarizes the total and unique vulnerabilities detected by each tool across the 240 tests. This analysis offers an overview of the performance and scope of each tool in identifying potential flaws. From Table 2, it is evident that the SAST tool *Visual Code Grepper* reports a considerably higher number and diversity of vulnerabilities compared to the SAST tool *Flawfinder*, highlighting its significantly broader coverage.

Figure 1 provides a more intuitive visualization of the differences between the total and unique detections presented by the tools Flawfinder and Visual Code Grepper. Note that the number of repeated vulnerabilities, represented by the difference between the two bars, shows values that are more similar between

Table 2: Summary of total vulnerabilities detected by each SAST tool.

Flawfinder		***Visual Code Grepper***	
Total	66,109	**Total**	101,124
Unique	49,261	**Unique**	82,224

the two tools, specifically 16,848 in Flawfinder and 18,900 in Visual Code Grepper.

For a more in-depth analysis, we examine the unique vulnerabilities categorized by severity level. As mentioned earlier, the SAST tool Flawfinder and the SAST tool Visual Code Grepper present their results differently: the former uses levels 1 to 5, while the latter uses levels 1 to 4. Table 3 illustrates the number of unique vulnerabilities, by severity level, for each of the analyzed tools. By analyzing Table 3, it can be seen that the SAST tool Flawfinder identifies a greater number of vulnerabilities with lower severity, while the SAST tool Visual Code Grepper identifies a greater number of vulnerabilities with higher severity.

Figure 2 shows the number of unique vulnerabilities by severity level for each of the analyzed tools. It is worth noting that the histogram shown in Figure 2 uses the original data for the two studied tools, which are on different scales. Nonetheless, it draws attention that the number of vulnerabilities with severity 3 found by the Flawfinder tool is quite small.

To facilitate the comparison of results obtained by the studied tools, we opted to normalize the values used to represent the severity levels. Normalization places the data on the same scale, which makes it easier to analyze the results. In this sense, we used the linear normalization approach, which maps the original values to a standardized scale ranging from 0 to 1. Below, we describe the linear normalization process in detail:

Let x be a value in the original table, and x_{norm} the corresponding normalized value. The formula for linear normalization is given by:

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Where $\min(x)$ represents the minimum value in the table and $\max(x)$ represents the maximum value.

Table 4 displays the relationship between the severity levels before and after the normalization process. Note that now the highest severity level in both tools is Level 1, and the lowest is Level 0. However, the SAST tool Flawfinder still maintains 5 severity levels, while the SAST tool Visual Code Grepper maintains 4 levels.

Figure 3 illustrates the number of unique vulnerabilities by severity level for each of the tools analyzed, using the normalized data. By analyzing Figure 3, it can be observed that the SAST tool Flawfinder identifies a greater number of vulnerabilities with lower severity, while Visual Code Grepper identifies a greater number of vulnerabilities with higher severity.

Now, we can analyze the vulnerabilities found according to the CWEs. As mentioned earlier, the SAST tool Flawfinder already provides two corresponding CWEs for each vulnerability detected in its results. On the other hand, the SAST tool Visual Code Grepper does not directly provide this categorization. To address this limitation, we listed all the vulnerabilities detected by Visual Code Grepper and manually assigned two CWEs to each of them. Then, we computed the number of detections for each CWE across the analyzed SAST tools.

Table 5 illustrates the number of vulnerabilities found by CWE for each of the tools investigated. Note that out of all 32 CWEs listed, the SAST tool *Visual Code Grepper* did not identify vulnerabilities for CWEs 785 and 829, confirming the extensive coverage of this tool. Additionally, Visual Code Grepper shows a lower number of detections compared to Flawfinder for CWEs 119, 120, 126, 134, 190, 250, 327, 362, 377, 676, 732, and 807 (twelve in total). Next, we briefly discuss some of these CWEs.

CWE-785: Use of Out-of-Bounds Pointer Arithmetic: Occurs when pointer arithmetic results in addresses outside the boundaries of memory allocated for an object. This can cause unexpected behavior, data corruption, segmentation faults, and exploitation possibilities by attackers (MITRE CWE).

CWE-829: Improper Handling of File Names or Paths: Involves insecure handling of file names or paths, allowing attackers to access or modify unauthorized files (MITRE CWE).

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer: Occurs when software performs operations on a memory buffer but reads or writes to a memory outside the intended buffer bounds. This can lead to memory corruption, arbitrary code execution, and denial of service (DoS).

Visual Code Grepper stands out for reporting a higher number of vulnerabilities compared to Flawfinder. This characteristic can be attributed to a more comprehensive approach in detecting potential security issues in the source code. Additionally, VCG tends to present more results categorized with higher criticalities, indicating a focus on more critical vulnerabilities. However, the tool does not

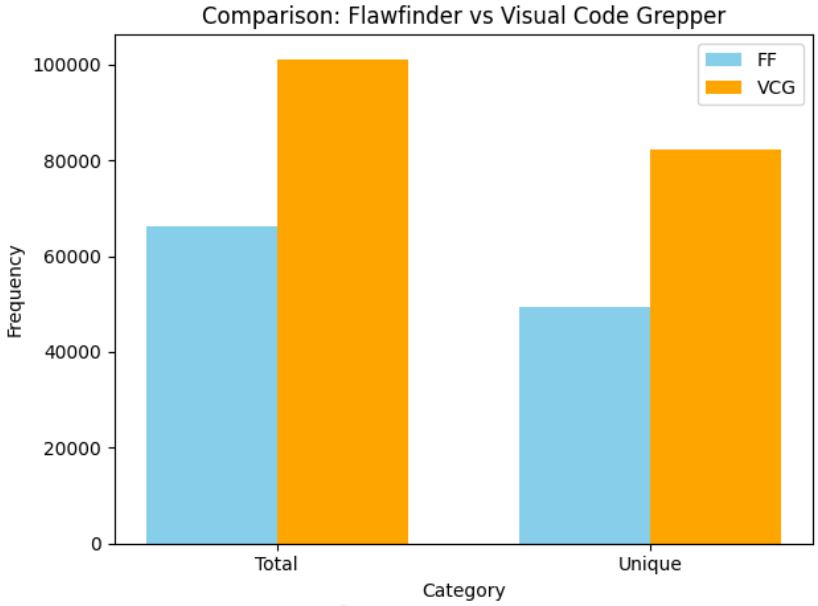

Figure 1: Total and unique vulnerabilities detected by each SAST tool.

Table 3: Distribution of unique vulnerabilities by severity levels for each tool.

Flawfinder		***Visual Code Grepper***	
Severity Level	***Vulnerabilities***	***Severity Level***	***Vulnerabilities***
1	12,569	**1**	464
2	24,541	**2**	309
3	2,164	**3**	54,688
4	9,838	**4**	26,763
5	149	**5**	-

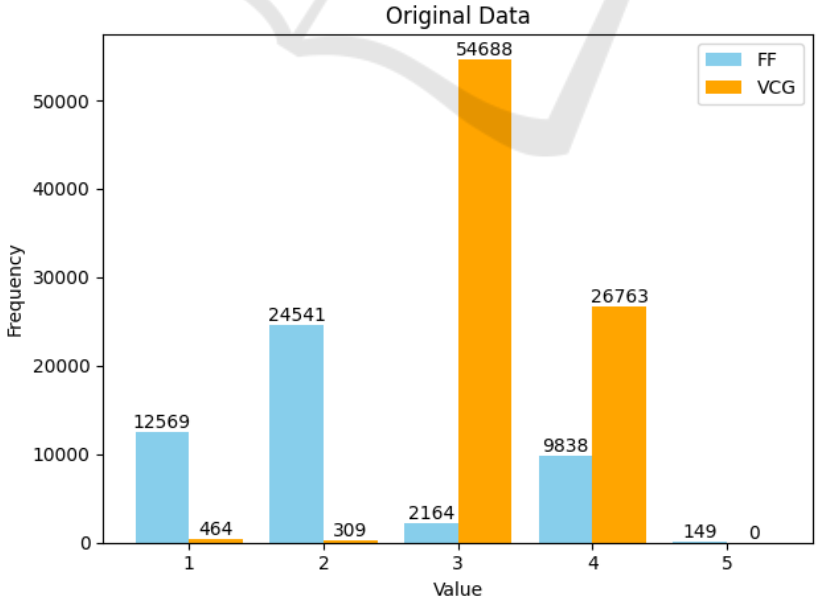

Figure 2: Unique vulnerabilities by severity level per SAST tool.

Table 4: Values for each severity level after normalization.

Flawfinder		*Visual Code Grepper*	
Original Level	***Normalized Level***	***Original Level***	***Normalized Level***
1	0	**1**	0
2	0.25	**2**	0.33
3	0.5	**3**	0.67
4	0.75	**4**	1
5	1	**5**	-

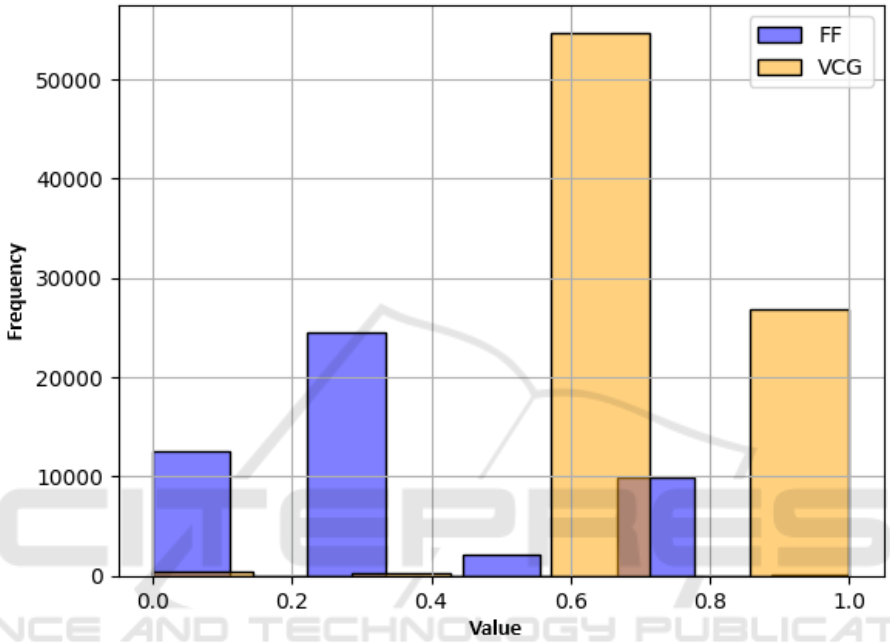

Figure 3: Unique vulnerability counts by severity levels for each tool (After Normalization).

categorize vulnerabilities according to the Common Weakness Enumeration (CWE). On the other hand, the SAST tool Flawfinder adopts a more balanced approach, reporting fewer vulnerabilities but distributed more evenly in terms of criticality. A notable advantage of Flawfinder is its greater versatility in searches and results, as it categorizes vulnerabilities according to the Common Weakness Enumeration (CWE). Furthermore, the capability of Flawfinder to be fully executed via the command line facilitates its integration into automated pipelines, making it more practical for incorporation into a software development pipeline.

5 CONCLUSIONS AND FUTURE WORK

In this paper, our goal was to investigate the feasibility of using open-source and standalone SAST tools for detecting security vulnerabilities in C/C++ projects. To achieve this objective, we conducted an empirical study in which we examined 30 C/C++ projects using two open-source and standalone SAST tools: Flawfinder (FF) and Visual Code Grepper (VCG). Additionally, this study was based on guidelines provided by the Common Weakness Enumeration (CWE) and the Open Web Application Security Project (OWASP) to identify and analyze the vulnerabilities present in the source codes available in these repositories. The results demonstrate the potential of using open-source and standalone SAST tools as a way to assess the security risks of a software product without the need to manually review all findings. Additionally, we observed that there are CWEs reported only by the Flawfinder SAST tool, as well as others reported solely by the Visual Code Grepper SAST tool. This highlights the need to use more than one SAST tool in the development process, to cover different perspectives in the testing process (Kleidermacher and Kleidermacher, 2012).

As future work, we intend to expand the number of repositories, including different programming lan-

Table 5: Quantities of vulnerabilities found by CWE.

CWE	FF	VCG	CWE	FF	VCG	CWE	FF	VCG
20	-	70	***248***	-	1	***480***	-	19.610
77	-	6	***250***	18	8	***676***	242	3
78	-	6	***327***	1.351	7	***681***	-	19.610
114	-	22	***362***	5.084	27	***690***	-	53
119	10.588	630	***367***	692	1.063	***703***	-	1.048
120	31.108	1.300	***377***	216	50	***732***	54	8
121	-	57	***396***	-	1	***772***	-	8
122	-	349	***401***	-	5.784	***785***	32	-
126	4.791	66	***404***	-	5.723	***807***	534	36
134	1.983	176	***427***	-	22	***829***	76	-
190	1.412	16	***432***	-	50			

guages and exploring a broader range of projects. Furthermore, we will incorporate new SAST tools with distinct approaches. Additionally, we plan to compare SAST, DAST, and IAST tools to provide a more comprehensive view of vulnerabilities at different stages of the development lifecycle. We also want to investigate the occurrence of false positives and negatives. Finally, we will apply the SAST tools in real production environments to gather feedback from developers and adjust/adapt the tools used.

ACKNOWLEDGMENTS

This work was partially funded by Lenovo as part of its R&D investment under the Information Technology Law. The authors would like to thank CNPq (316729/2021-3) and LSBD/UFC for the partial funding of this research.

REFERENCES

Ahmed, A. (2019). Devsecops: Enabling security by design in rapid software development. Master's thesis.

Baca, D., Carlsson, B., and Lundberg, L. (2008). Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88.

Baldissera, A. (2001). Pesquisa-ação: uma metodologia do "conhecer" e do "agir" coletivo. *Sociedade em Debate*, 7(2):5–25.

de França, R. P. and da Silva, V. B. (2022). Devsecops-integração da segurança contínua em pipelines devops: um estudo de caso. In *Anais Estendidos do XXII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 272–285. SBC.

Dias, R. d. C. W. B. et al. (2023). Avaliação comparativa das metodologias na gestão de projetos.

Esposito, M., Falaschi, V., and Falessi, D. (2024). An extensive comparison of static application security testing tools.

Fagan, M., Megas, K. N., Scarfone, K., and Smith, M. (2020). Atividades fundamentais de cibersegurança para fabricantes de dispositivos iot.

Kleidermacher, D. and Kleidermacher, M. (2012). *Embedded systems security: practical methods for safe and secure software and systems development*. Elsevier.

Macedo, M. H. B. and Salgado, E. G. (2015). Gerenciamento de risco aplicado ao desenvolvimento de software. *Sistemas & Gestão*, 10(1):158–170.

Parizi, R., Moreira, M., Couto, I., Marczak, S., and Conte, T. (2020). A design thinking techniques recommendation tool: An initial and on-going proposal. In *Proceedings of the XIX Brazilian Symposium on Software Quality*, pages 1–6.

Rahul, B., Prajwal, K., and Manu, M. (2019). Implementation of devsecops using open-source tools. *International Journal of Advance Research, Ideas and Innovations in Technology*, 5(3).

Santos, L. D. V. and de Santana Oliveira, C. V. (2017). *Introdução à garantia de qualidade de software*. Cia do eBook.

Shi, Z., Matyunin, N., Graffi, K., and Starobinski, D. (2024). Uncovering cwe-cve-cpe relations with threat knowledge graphs. *ACM Trans. Priv. Secur.*, 27(1).

Wheeler, D. A. (2015). Secure programming howto. *Walters Art Museum in Baltimore, Maryland*.

Woody, C., Chick, T., Reffett, A., Pavetti, S., Laughlin, R., Frye, B., and Bandor, M. (2020). Devsecops pipeline for complex software-intensive systems: Addressing cybersecurity challenges. *The Journal on Systemics, Cybernetics and Informatics: JSCI*, 18(5):31–36.