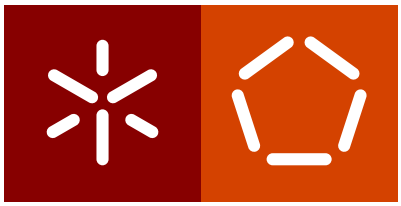
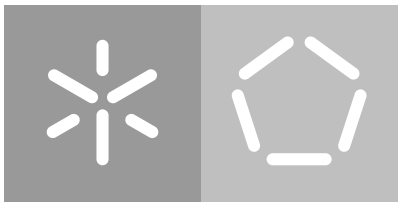


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Alexandra de Barros Reigada

Generic SAST tool Comparer

Academic Year 2021/2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Alexandra de Barros Reigada

Generic SAST tool Comparer

Master dissertation
Integrated Master Degree in Informatics Engineering

Dissertation supervised by
Pedro Rangel Henriques
Nuno Oliveira

Academic Year 2021/2022

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Alexandra de Barros Reigada

ACKNOWLEDGMENTS

Over the last five years, I had the support of some important people that were always by my side and were essential to the achievement of my integrated master's degree in software engineering, whom I would like to thank.

First of all, to my supervisors: Professor Doutor Pedro Rangel Henriques thank you for accepting to guide this project, for your weekly support, your motivation and enthusiasm for my accomplishments, your knowledge and advisory, throughout this year; Doutor Nuno Oliveira thank you for your constant availability, all the guidance, for sharing your vision with me and finally for giving me the opportunity to be an intern at Checkmarx which contributed for the development of this project and my personal growth.

To all my family, thank you for always being present, for your efforts, unconditional love and for never letting me give up. I couldn't do it if you were not by my side.

To my MIEI colleagues, thank you for sharing knowledge, for the mutual help and for your friendship. This journey wouldn't be so happy without you.

To my friends from Ponte de Lima, thank you for your support and for always being there, especially in the rougher times.

Thank you, Tuna de Medicina da Universidade do Minho, for all the wonderful moments that we shared, for the motivation, for helping me forget about my issues and for bringing excitement and positivity into my academic life.

To my team Adamastores from Checkmarx, thank you for the kind way you welcomed me to the company and guidance in my first work experience. Without your flexibility, support and understanding of my academic status, I wouldn't have accomplished this important step.

Finally, thank you Checkmarx AppSec for your availability and for making a valuable contribution to this project.

Thank you everyone, you will always be in my heart
Alexandra

ABSTRACT

Cybernetic attacks are a genuine concern today that can compromise the integrity of any person, organization, or business. Every day, new incidents are publicized that demonstrate the true extent of the harm that cyber criminals may wreak. Sensitive data exposure, identity theft, service malfunctioning are just a few of the most typical dangers, which can result in financial loss or damage to a company's reputation in many circumstances. There are many mechanisms and technologies used by these companies to identify vulnerabilities in applications. The most popular technology used to detect vulnerabilities is SAST (Static Application Security Testing) as it focus on the detection of vulnerabilities at the early stages of software development. However, these tools only analyze source code and that brings a big problem associated: wrong detections (false positive results) and some real vulnerabilities not reported (false negative results), adding that depending on what techniques used by each product, this number may change and the content of the results also changes. With that said, SAST solution providing companies would benefit from a system where it would be possible to compare their SAST results against results of adverserial products. This would allow them to understand their flaws and opportunities to improve. In the context of the above, Checkmarx proposes the development of a system to compare SAST tool results providing insight on how one such tool falls behind its competitor and how it can be improved to match the exposed gap. This is the main topic of this Master's Thesis dissertation. An exhaustive study of SAST tools, their classification according to a set of predefined dimensions and platforms that compare these tools were the starting point to make the system generic, innovative and able to compare the great number of SAST tools in the market. SAST Tool Comparer has been developed following an architecture that fulfills the expected and proposed functionalities: it reads scan reports from several SAST tools, either open-source and commercial, obtains results from open-source SAST tools directly scanning from the application, compares results taking into account multiple parameters, displays important statistics to understand the comparison and also provides the configuration and introduction of new tools as well as converters to filter the necessary results for an efficient comparison. In this document, each of these features will be presented from the implementation until the final result, as well as some comparisons between SAST tools and possible inferences resulting therefrom.

Key-Words: vulnerability detection, static code analysis, SAST tools, SAST-tools comparison

RESUMO

Os ataques cibernéticos são uma preocupação genuína hoje em dia que podem comprometer a integridade de qualquer pessoa, organização ou empresa. Todos os dias, novos incidentes são divulgados que demonstram a verdadeira extensão dos danos que os criminosos cibernéticos podem causar. Exposição de dados confidenciais, roubo de identidade, mau funcionamento do serviço são apenas alguns dos perigos mais comuns, que podem resultar em perdas financeiras ou danos à reputação de uma empresa em muitas circunstâncias. Existem muitos mecanismos e tecnologias utilizados por essas empresas para identificar vulnerabilidades em aplicações. A tecnologia mais popular é o SAST (Static Application Security Testing), por se concentrar na detecção de vulnerabilidades nas fases iniciais do desenvolvimento de software. No entanto, estas ferramentas apenas analisam código-fonte e esse facto contém um problema associado: identificações erradas (falsos positivos) e algumas vulnerabilidades que existem e não são descobertas (falsos negativos), acrescentando que dependendo das técnicas utilizadas por cada produto, esse número e o conteúdo dos resultados podem mudar. Dito isto, as empresas que providenciam este tipo de ferramenta podem beneficiar de um sistema onde seria possível comparar os seus resultados SAST com os de produtos concorrentes. Isto permite-lhes entender as suas falhas e oportunidades para melhorar. Dentro deste contexto, a Checkmarx propôs o desenvolvimento um sistema que compara os resultados da utilização de ferramentas SAST, fornecendo uma análise sobre o seu comportamento quando testadas com a concorrência. Um estudo exaustivo das ferramentas SAST, a sua classificação de acordo com um conjunto de dimensões pré-definidas e plataformas que comparam essas ferramentas foram o ponto de partida para tornar o sistema genérico, inovador e capaz de comparar um número substancial de ferramentas presentes no mercado. SAST Tool Comparer foi elaborado seguindo uma arquitetura que cumpre as funcionalidades esperadas e propostas: lê *scan reports* de várias ferramentas SAST tanto *open-source* como comerciais, obtém resultados de ferramentas SAST *open-source* diretamente da aplicação, compara resultados tendo em conta vários critérios e apresentando estatísticas importantes para perceber a comparação e ainda disponibiliza a configuração e introdução de novas ferramentas bem como conversores para filtrar os resultados necessários para uma comparação eficiente. Neste documento serão apresentadas cada uma destas funcionalidades desde a implementação até ao resultado final bem como algumas comparações entre ferramentas SAST e possíveis inferências daí resultantes.

Palavras-Chave: deteção de vulnerabilidades, análise estática de código, ferramentas SAST, comparação de resultados do SAST

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

AST Abstract Syntax Tree.

C

CWE Common Weakness Enumeration.

D

DAST Dynamic Application Security Testing.

H

HTTP Hypertext Transfer Protocol.

I

IAST Interactive Application Security Testing.

L

LDAP Lightweight Directory Access Protocol.

S

SAST Static Application Security Testing.

SDLC Software Development Life Cycle.

SQL Structured Query Language.

SVM Suport Vector Machine.

X

XML Extensible Markup Language.

XSS Cross-Site Scripting.

XXE XML External Entities.

1

INTRODUCTION

Nowadays, information security is one of the most relevant areas in the business world.

The need to obtain data and essential information for organizations explains the importance of information security procedures that prevent risks.

The invasion of business systems, sensitive data exposure, identity theft or any other cyber attack are malicious actions very present in today's world that damage not only the company reputation but also cause financial losses.

Since nowadays almost everyone has a device with an Internet connection, cyber security has become a major concern for software companies. It is essential to have the capability to identify and correct pieces of vulnerable code in order to prevent the software from being compromised, maintain the clients safety and assure the security of company data.

A vulnerability is a weakness in an application that allows an attacker to cause harm to the stakeholders of that application. Stakeholders include the application owner, application users, and other entities that rely on the applications[1].

The Open Web Application Security Project (OWASP)[2] defines Static Application Security Testing (SAST) tools as those that can help find security vulnerabilities in the source code without compiling and running the program. Such tools detect and classify the vulnerability warnings into one of many types (e.g., input validation and representation)[Aloraini et al. (2019)].

Finding coding errors early in the development life cycle saves the time and money of catching them during production and makes sure the code is written securely as it is created.

SAST tools offer organizations a number of benefits such as the ability to find vulnerabilities without the need for compiling or running code or the coverage of different programming languages, maning it is easy to identify common vulnerabilities like buffer overflows, Cross-site scripting problems and SQL injection flaws. Moreover, after they find an error, they clearly identify source files, line numbers, subsections of lines containing

1 Avaliable at: https://owasp.org/www-community/vulnerabilities/, accessed in October 2021.
2 Available at: https://owasp.org/www-community/Source_Code_Analysis_Tools, accessed in October 2021.

errors and even complete flows of data from code constructs representing possible inputs to those representing sinks of the identified vulnerabilities[3].

SAST tools constitute, thus, one of the most effective approaches for finding software vulnerabilities early in the development life-cycle. However, static analysis has a few limitations due to the lack of runtime information. In SAST tools this lack of information may severely impact the quality of the findings, namely their accuracy.

1.1 MOTIVATION

The accuracy of the findings is always a high requirement in SAST tools. SAST tools are known to produce false positives (finding a vulnerability where it doesn't exist and hence, do not require developers' effort for fixing) and false negatives (not finding a vulnerability where it exists) and accuracy is impacted by these findings.

Low accuracy (this is, a great amount of such wrong results) is most of the times, what raises more concerns to costumers, affecting the confidence on such tools and their adoption. Moreover, a high rate of false warnings makes developers lose interests in detecting results [Tow] specially with larger projects, where the number of warnings produced by a tool can be high and can "outweigh" the true positives[Johnson et al. (2013)]. One possible approach to find missing results, is usually to rely on on-purpose vulnerable projects, knowing however that these are not real-life projects. Other approach could be to compare the findings with findings obtained by other SAST tools. Therefore, motivated by the above-mentioned challenges that developers and costumers may encounter when using SAST tools, this project aims at the development of a system that compares SAST results hoping to understand how to improve the performance of the used techniques.

1.2 OBJECTIVES

This Master's Thesis objectives are the following:

- Analysis, description, and comparison of various SAST tools.
- Creation of a generic and extensible system capable of comparing existing SAST tools and integrating new ones. The output must indicate the differences in the results of both tools.
- Apply the created system to compare several SAST tools and assess their outputs.

3 Avaliable at: https://www.softwaresecured.com/top-sast-tools-for-developers/, accessed in October 2021.

1.3 RESEARCH HYPOTHESIS

With this master's work, it is intended to prove that it is possible to automatically compare SAST tools results by directly scanning with such tools - in case they are free or open source - or considering the output findings – in case of commercial, aiming at understanding gaps in their findings, in order to improve the accuracy of a specific one.

1.4 RESEARCH METHOD

The methodology used to achieve the objectives will be the following:

- Research of SAST tools available in the market, commercial or open source;
- Analysis and description of the SAST tools referred above, how do they express their output, what input accept, what technologies are used to produce the ouput, what is the mode of execution;
- Bibliographic study to deeply understand the state of the art in the area of SAST tools comparison platforms;
- Comparison of results obtained and construction of a table;
- Development of a generic and extensible platform capable of integrating the SAST tools in order to indicate what they have in common, if one of them spots more vulnerabilities than the other and if so, what upgrade should be added to close the identified gaps;
- Test this platform with Checkmarx SAST tool, and other.
- Discussion of results.

This is an iterative process. If the results achieved in one of the stages are not satisfactory, then it is important to go back to one of the previous stages and deepen the literature research or revise some project decisions.

1.5 DOCUMENT STRUCTURE

The dissertation is organized into eight chapters. In this chapter, the context, motivation and objectives of this project were presented. Chapter 2 presents some important definitions and clarifications about the subject that will be discussed. Chapter 3, "State of the Art", presents a summary of the most known SAST tools and related existing work on comparing

these tools. Chapter 4, "Proposed Approach", explains the strategy chosen, system architecture and chosen technologies to construct the system. Chapter 5 explains all the steps and decisions that were made to develop the system proposed that will allow for reach the objectives defined. Chapter 6 describes the application built, the usage flow and its functionalities. Chapter 7 validates the system developed, presenting three case-studies. Finally Chapter 8, the Conclusion, is dedicated to express conclusive thoughts about the project outcomes and future work that could improve this platform.

2

BACKGROUND

This chapter covers some main concepts that are important to better understand and follow the next chapters such as the definition of a code vulnerability and SAST.

2.1 CODE VULNERABILITY

A vulnerability is a flaw or error in the code of a system or device that, if it is exploited, might compromise the confidentiality, accessibility, and integrity of data stored there by allowing unauthorized access, elevating privileges, or denying services. An exploit is a piece of software or a program used to take advantage of this weakness.

There are some communities that help in identifying commonly occurring risks in applications [Pariwish Touseef (2019)] such as:

- System Administration, Networking, and Security (SANS)
  SANS is a private organization that provides training in cybersecurity and information security. This institute keeps a list of the 25 most dangerous software error types[1].

- Common Vulnerabilities and Exposure (CVE)[2]
  The MITRE organization created CVE in 1999 to find and classify software and firmware vulnerabilities. CVE offers a free lexicon to businesses to help them improve their cyber security. Security experts may acquire information about certain cyberthreats from various information sources using CVE identifiers, also known as CVE names or CVE numbers, by utilizing the same common term.

- Common Weakness Enumeration (CWE)[3] is a community-developed collection of software and hardware weakness types. It serves as a common language, a measuring stick for security tools, and as starting point for attempts to identify, mitigate, and avoid weaknesses.

1 Available at: https://www.sans.org/top25-software-errors/, accessed in July 2022.
2 Available at: https://cve.mitre.org/, accessed in July 2022.
3 Available at: https://cwe.mitre.org/data/, accessed in July 2022.

The Common Weakness Enumeration Top 25[4] is a list of the most widespread and critical weaknesses that are likely to lead to exploits, repairs, and lengthy lulls in development.

Simply said, CVE and CWE differ in that one addresses symptoms while the other addresses a root cause. The CVE is only a list of currently known problems with certain systems and products, whereas the CWE classifies different types of software vulnerabilities.

CVE may be used to maintain security controls for software assurance, however it is not as relevant as CWE is[5].

- Open Web Application Security Project (OWASP)
  The OWASP is a set of coding standards provided by a free online community established to deliver recommendations, processes, documentation, tools, and guidelines to develop secure software.

  The OWASP Top 10 is a standard awareness document for developers and web application security. Companies should adopt this document and start the process of ensuring that their web applications minimize these risks[6].

  Some of the most common security vulnerabilities present in the OWASP Top Ten Web Application Security Risks of 2021 are :

  1. **Broken Access Control**
     By enforcing policy, access control ensures that users stay inside the bounds of their specified permissions. Failures frequently result in the unauthorized exposure of information, the change or deletion of data, or the performance of business functions outside the user's scope.
  2. **Cryptographic Failures**
     This issue leads to the exposure of confidential application data such as passwords, patient health records, business secrets, credit card numbers, email addresses and other private user data. This happens when there is a poor or non-use of cryptographic algorithms.
  3. **Injection**
     In this type of attack, an attacker supplies untrusted input to a program. An interpreter will process this input as part of a command or query. This in turn modifies how that software is run, when the User-supplied data is not filtered, validated or sanitized.

4 Available at: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, accessed in July 2022.
5 Available at: https://www.parasoft.com/blog/what-is-cwe/, accessed in July 2022.
6 Available at: https://owasp.org/www-project-top-ten/, accessed in July 2022.

4. **Insecure Design**
   This new category refers to different weaknesses, expressed as absent or ineffective control design. A flawless implementation cannot remedy an unsafe design since, by definition, the necessary security safeguards were never developed to protect against certain attacks.
5. **Security Misconfiguration**
   These attacks take use of web applications' setup flaws. Many web applications come with important developer features that, if not disabled during live production, are risky.
6. **Vulnerable and Outdated Components**
   This happens when a program makes use of elements with known security flaws, including libraries or APIs. A developer must know the version of components being used, check for problems regularly, remove redundant and unnecessary dependencies.
7. **Identification and Authentication Failures**
   Failures in identification, authentication and session management are flaws in the apps' authentication methods.
8. **Software and Data Integrity Failures**
   Occurs when significant information and software updates are inserted into the delivery pipeline without first ensuring their authenticity. It is crucial to check installed packages, used libraries and dependencies and ensure that the data is authentic and has not been altered in any way.
9. **Security Logging and Monitoring Failures**
   This flaw exists when there are insufficient logging and monitoring procedures that can help identify some type of dangerous activity.
10. **Server-Side Request Forgery**
    This type of attack allows the attacker to view or modify internal resources by abusing server capabilities. By carefully choosing the URLs, the attacker may be able to read server configuration information like AWS metadata, connect to internal services like http-enabled databases, or send post requests to internal services that are not intended to be exposed. The attacker can provide or modify a URL that the code running on the server will read from or submit data to.

What constitutes a bug or a vulnerability depends on the programming languages and frameworks used. A programming issue in Java language looks very different from bad C or Swift code. Therefore, static code analysis is programming language dependent.

2.1.1 *CWE*

The Common Weakness Enumeration (CWE) is a category system for hardware and software weaknesses and vulnerabilities. It is supported by a community project whose objectives are to comprehend hardware and software defects and develop automated tools that may be used to find, correct, and prevent those problems [7].

This list contains a specific and succinct definition for each common weakness type as well as examples on how it can occur, thus this community makes an effort to ensure that every item in the list is adequately described and differentiated.

All of these weaknesses are connected by hierarchical relationships so they can be navigated by a specific point of view.

Each CWE has the following structure as seen in Figure 1:

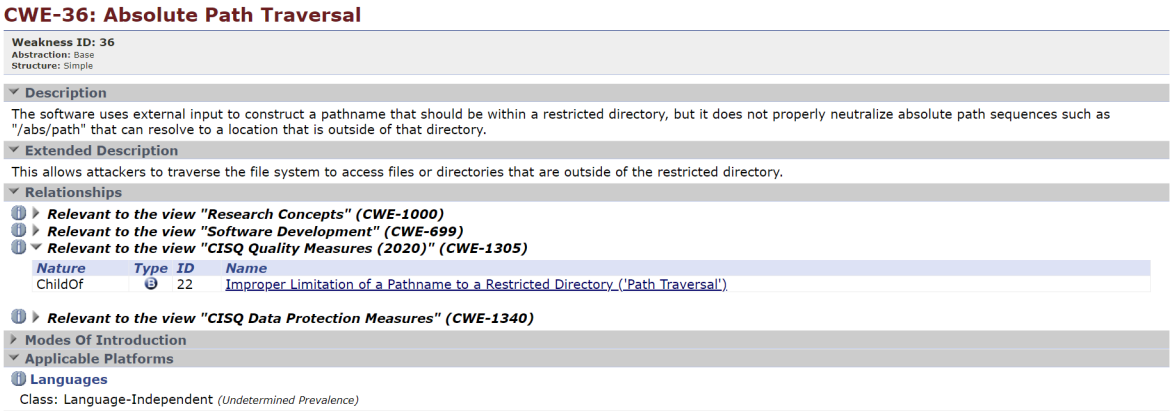


Figure 1: CWE Structure - ID, name, description and relationships.

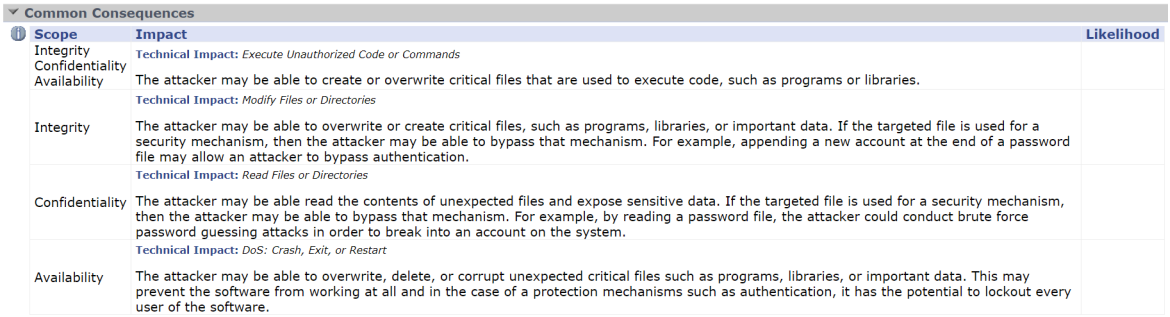
It has a name and an identifier which are unique. Next there is a description for this issue, how it can occur, how can an attacker take advantage of this and what are the consequences.

There is a section dedicated to its relationships according to certain views. These interactions - defined as ChildOf, ParentOf, MemberOf - give insight to related weaknesses that could exist at higher and lower levels of abstraction. Also, defined to identify comparable CWEs that the user may wish to investigate are relationships like PeerOf and CanAlsoBe.

For example, CWE-36 is a child of CWE-22 which refers to *Improper Limitation of a Pathname to a Restricted Directory*.

7 Available at: https://cwe.mitre.org/, accessed in July 2022.

The applicable platforms table shows possible areas where the given weakness could appear. These may be for certain identified languages, operating systems, architectures, paradigms, technologies, or a class of such platforms. Each one of these can have a frequency of this appearance. For this case in special, Absolute Path Traversal can occur in any language. Nonetheless, there are some CWE that are related to certain languages. For example, CWE-125 - Out-of-bounds Read usually appears on C or C++ projects.

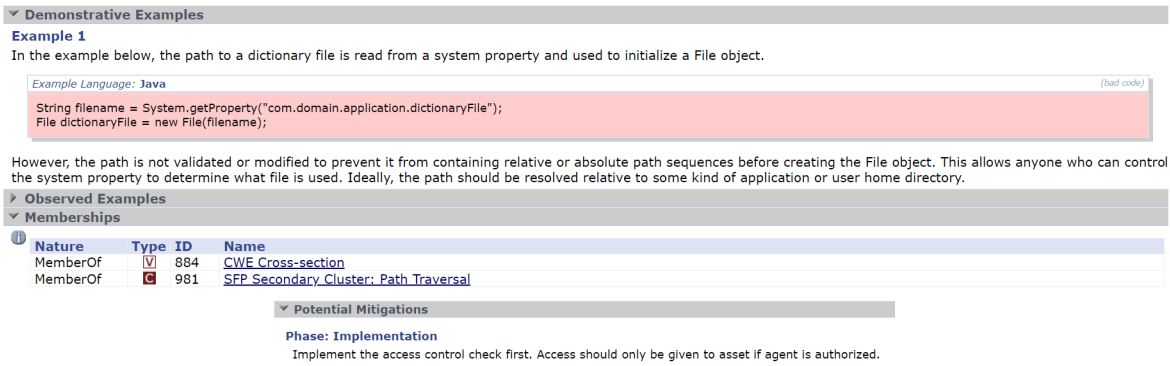


Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	**Technical Impact:** *Execute Unauthorized Code or Commands* The attacker may be able to create or overwrite critical files that are used to execute code, such as programs or libraries.	
Integrity	**Technical Impact:** *Modify Files or Directories* The attacker may be able to overwrite or create critical files, such as programs, libraries, or important data. If the targeted file is used for a security mechanism, then the attacker may be able to bypass that mechanism. For example, appending a new account at the end of a password file may allow an attacker to bypass authentication.	
Confidentiality	**Technical Impact:** *Read Files or Directories* The attacker may be able read the contents of unexpected files and expose sensitive data. If the targeted file is used for a security mechanism, then the attacker may be able to bypass that mechanism. For example, by reading a password file, the attacker could conduct brute force password guessing attacks in order to break into an account on the system.	
Availability	**Technical Impact:** *DoS: Crash, Exit, or Restart* The attacker may be able to overwrite, delete, or corrupt unexpected critical files such as programs, libraries, or important data. This may prevent the software from working at all and in the case of a protection mechanisms such as authentication, it has the potential to lockout every user of the software.	

Figure 2: CWE Structure - Common consequences.

The many different repercussions linked to the vulnerability are listed in *Common Consequences* table as seen in Figure 2 *Scope* identifies the application security area that is being compromised, while *Impact* describes the negative technical impact that comes up if an attacker succeeds in exploiting this flaw. *Likelihood* gives details about how likely a particular consequence will occur compared to the others on the table. For instance, a vulnerability may have a high chance of being used to achieve a certain impact but a low chance of being used to achieve a different impact.



Demonstrative Examples

Example 1

In the example below, the path to a dictionary file is read from a system property and used to initialize a File object.

Example Language: **Java** (bad code)

```
String filename = System.getProperty("com.domain.application.dictionaryFile");
File dictionaryFile = new File(filename);
```

However, the path is not validated or modified to prevent it from containing relative or absolute path sequences before creating the File object. This allows anyone who can control the system property to determine what file is used. Ideally, the path should be resolved relative to some kind of application or user home directory.

Observed Examples

Memberships

Nature	Type	ID	Name
MemberOf	V	884	CWE Cross-section
MemberOf	C	981	SFP Secondary Cluster: Path Traversal

Potential Mitigations

Phase: Implementation

Implement the access control check first. Access should only be given to asset if agent is authorized.

Figure 3: CWE Structure - Examples, memberships and mitigations.

These last listings as seen in Figure 3 describe some examples on which this issue occurs as well as the views in which this CWE is present (memberships). Finally there is a table of

possible mitigations that suggest a prevention for a likely attack.

2.2 INTRODUCTION TO STATIC APPLICATION FOR SECURITY TESTING

Static Application Security Testing, also known as white-box testing, is a software testing methodology designed for inspecting and analyzing application source code that could disrupt the availability and integrity of an application's service. This happens without the code being executed, in the initial stages of development[8].

Therefore, SAST tools prevents security-related issues from being considered an afterthought. This give developers real-time feedback as they code helping them make necessary improvements fast and efficiently without having to check through the code manually. SAST scans are based on a set of predetermined rules that define the coding errors in the source code that need to be addressed and assessed[9]. SAST tools detect bugs, vulnerabilities and improve code quality [Nguyen-Duc et al. (2021)].

The SAST service intends to enable developers to design and deliver secure code by integrating the SAST tools into existing development and/or delivery pipeline procedures, which helps developers uncover and fix vulnerabilities even before a project reaches the testing phase.[Li (2020)]. With that said, SAST tools can integrate with DevSecOps[10] CI/CD pipelines, run in an IDE or by command line. Developers can also customize reports, these can be exported offline and tracked using dashboards.

However, the current status of SAST tools is susceptible to the following drawbacks:

- A large number of false positives and negatives reported, making it difficult to determine that a security issue is actually a vulnerability. As a result, developers will have to put in more work to manually discover and address errors.
- It is still challenging to automatically detect a few types of security flaws (such as authentication issues, access control problems, and unsafe cryptography usage).
- It is difficult to find bugs in third-party libraries, customizations, and frameworks because they aren't always reflected in the code.
- The incapability of reviewing compiled source code and detect business logic flaws[Li (2020)].

8 Available at: https://www.synopsys.com/blogs/software-security accessed in November 2021.
9 Available at: https://owasp.org/www-community/Source_Code_Analysis_Tools, accessed in November 2021.
10 Available at: https://www.devsecops.org/, accessed in November 2021.

2.3 SUMMARY

This chapter highlighted key elements surrounding SAST, including the concept of code vulnerability and what CWE and SAST are.

3

STATE OF THE ART

This chapter will present the existing tools in the market, some articles and platforms that compare these tools and what type of work has been done around this topic.

3.1 EXISTING SAST TOOLS

In this section, it is presented some of the existing tools in the market, commercial and open-source, understand their mode of execution, what type of output is produced, type of analysis and scanning.

It is divided in two subsections: open source and commercial tools. Open source products usually support one language only and use simplistic algorithms that are incapable of detecting specific flaws. Open source tools have far smaller rule bases, weaker interfaces, and integration functionality as compared to commercial alternatives. Commercial solutions use complex specific algorithms, have extensive rule bases, support a variety of programming languages, and provide a comprehensive user interface and integration (e.g., plugins or APIs).

3.1.1 *Open source tools*

Flawfinder

Flawfinder is a tool that detects vulnerabilities in C/C++ projects by using pattern-matching and is executed by command-line. This tool produces a report in HTML or CSV format or prints it in the command-line. This report lists all files that were scanned and all the vulnerabilities found, ordered by severity. For each vulnerability, Flawfinder provides its location, the severity score, its type, the vulnerable function, a short description, a link to the CWE page of the vulnerability, and a proposed fix which is often the name of a safe function that can be used instead of the vulnerable one. Moreover, the report shows the analysis summary, which contains statistical data about the scan, such as with the number

of files scanned and the number of errors reported[Smith et al. (2020)].

NodeJsScan

NodeJsScan supports many languages such as Java, C++, C, VB, PHP, PL/SQL. It has a command line interface for easy integration with DevSecOps CI/CD pipelines. It produces results in JSON. A configuration file is available for each language which can be modified for customized searches[har]. NodeJsScan comes with a collection of security rules defined in the rules.xml file, which includes tags that identify different sorts of vulnerabilities as well as rules to match vulnerabilities in the project's source. String comparison, Regex comparison, Template comparison, Multi-Match Regex comparison, Dynamic Regex comparison, and Missing Security Code are the six categories of rules.

Semmle

Semmle Inc[1] is a platform for code analysis owned by GitHub. Semmle developed LGTM, a continuous code analysis tool focused on identifying vulnerabilities in software systems that has Semmle QL in its core, a query language and code search engine that enables code analysis to detect and eliminate security vulnerabilities. QL employs variant analysis, a technique commonly used by security experts. After a vulnerability is discovered, security experts examine the remainder of the code base for similar issues. QL automates and extends this process across various several code bases, allowing developers to design queries that can be shared and reused. Semmle claims that their solutions have identified hundreds of vulnerabilities, including over 100 CVEs in open source projects[2].

Brakeman

Brakeman is a vulnerability scanner specifically designed for Ruby on Rails applications. It produces a HTML or Generic Issue Import Format output and provides a quick look into the code which raised the warning. It runs by command line and there is a plugin available for Jenkins/Hudson[3],[4]

1 Available at: https://github.com/Semmle, accessed in July 2022.
2 Available at: https://www.infoq.com/news/2019/09/github-semmle-vulnerabilities/, accessed in July 2022.
3 Available at: https://brakemanscanner.org/, accessed in December 2021.
4 Available at: https://github.com/presidentbeef/brakeman

Findbugs

Findbugs[5] is an open source project that detects 125 types of security vulnerabilities in Java programs. Findbugs report and classify the bugs and vulnerabilities found into four rankings: scariest, scary, troubling, and of concern. It contains a bug description, examples of similarly vulnerable code and how to fix it, links to useful information on this vulnerability, tool-specific information and it can be customized so only a subset of the categories are reported on. It also provides a "Navigation" panel that contains a trace of the vulnerability and offers a few quick fixes[Johnson et al. (2013)],[har],[Smith et al. (2020)]. Findbugs can be executed by command line and there is also an Eclipse plugin[Smith et al. (2020)].

RIPS

RIPS (Research and Innovation to Promote Security)[6] is the most popular static code analysis tool for PHP code and can detect more than 80 vulnerabilities. This tool tokenizes PHP code (lexical analysis) based on PHP's tokenizer extension and performs semantic analysis to build a program model. RIPS provides an integrated code audit architecture in addition to a structured report of discovered vulnerabilities. It provides a web interface from which the user can configure and launch scans, and consult the results. Vulnerabilities are categorized into files and then sorted by type of vulnerability. Along with the problematic code, RIPS provides a brief summary of each vulnerability. The help view, when available, explains vulnerabilities in greater depth and may recommend fixes.[Smith et al. (2020)]. False positives are a drawback of the open-source version due to the lack of an abstract syntax tree or control-flow graph. False negatives can occur due to a lack of support for object-oriented PHP programming. There is a commercial version of RIPS that can analyze not only PHP but also Java code. This one, on the other hand, makes use of control-flow graphs and abstract syntax trees. RIPS is accessible as both on-premises software and Software-as-a-Service (SaaS)[7].

JsHint

JsHint is a SAST tool that detects vulnerabilities in JavaScript programs. It is designed to help developers write complex programs without worrying about typos and language errors[har]. There is an online version which is accessible through the official website and a

5 Avaliable at: ttp://findbugs.sourceforge.net/, accessed in December 2021.
6 Avaliable at: ttp://rips-scanner.sourceforge.net/, accessed in December 2021.
7 Available at: https://en.wikipedia.org/wiki/RIPS, accessed in December 2021.

command-line version, distributed as a Node.js module [8].

CodeWarrior

CodeWarrior can find security vulnerabilities in applications written in C, C#, PHP, Java, Ruby, ASP, and JavaScript and is available for Linux, OX, BSD, and MacOS. It can be executed after downloading it and constructing it with "make". Furthermore, although it is a web application, Apache is not required to operate it. The program is known for having a low percentage of false positives[har].

SonarQube

SonarQube is one of the most common open-source static code analysis tools for measuring quality aspects of source code, including vulnerability. The Community Edition includes static code analysis for around 15 languages as well as vulnerability and bug detection, code smell tracking, technical debt review with remediations, code quality history and metrics, CI/CD integration, and the ability to extend functionality further with over 60 community plugins. The Developer Edition includes all of the above features plus support for 22 languages (ABAP, C, C++, CSS, Flex, HTML, Go, JavaScript, Java, Objective-C, Kotlin, PL/SQL, PHP, C, Python, Ruby, Scala, Swift, T-SQL, VB.Net, TypeScript, and XML), as well as real-time notifications in the IDE as part of SonarLint smart notification and pull request decoration. SonarQube implements two fundamental approaches to check for issues in source code:

- Syntax trees and API basics: Before running any rules, a code analyzer parses the given source code file and produces the syntax tree. The structure is used to clarify the problem as well as determine the strategy to use when analyzing a file.
- Semantic API: In addition to enforcing rules based on data provided by the syntax tree, SonarQube provides more information through a semantic representation of the source code. However, for the time being, this model only works with Java source code. This semantic model offers data on each symbol that is changed.[Nguyen-Duc et al. (2021)].

3.1.2 *Commercial tools*

DeepCode

8 Available at: https://www.methodsandtools.com/tools/jshint.php, accessed in December 2021.

DeepCode is a SAST tool that uses AI to find vulnerabilities. It supports Java, JavaScript, Python, TypeScript, and C/C++ code. For open source software and commercial teams of up to 30 developers, the bot is free. DeepCode's machine learning module analyzes a large repository of modifications made by developers as they work on a wide range of projects. DeepCode is able to provide developers a potential solution to the problem they are working on by learning from this repo, as well as spot faults that have previously occurred[9]. Developers can connect DeepCode with their GitHub, BitBucket, or GitLab accounts or directly within their IDE. DeepCode immediately starts reviewing each commit and identifies issues without any required configuration[10].

Reshift Security

Reshift Security is a SAST tool that supports Java and JavaScript. It integrates with Github, Bitbucket, and Gitlab where it can simply sync projects and run scans on every build. With the focus being a tool built for developers, Reshift offers "automated fixes" where suggested fixes are listed and developers can simply accept to create a pull request. Because of that, Reshift is currently known for being a seamless tool that developers can use and don't have to interact with it to remediate issues[har]. Reshift can be used as a plugin for IntelliJ and there is also a VS Code extension.

CodeSonar

CodeSonar[11] is a tool that performs a whole-program, interprocedural analysis on C/C++ code and identifies complex programming bugs. Similar to a compiler, CodeSonar does a build of the code, but instead of creating object code it creates an abstract representation of the program. After the individual files are built, a synthesis phase combines the results into a whole-program model. The model is symbolically executed and the analysis keeps track of variables and how they are related. Warnings are generated when anomalies are encountered. CodeSonar does not need test cases and works with the existing build system[Ivo Gomes (2009)] CodesSonar produces a report in SARIF (Static Analysis Interchange Format).

AppScan

AppScan[12], formerly by IBM, is a SAST designed for web applications that uses machine learning which allows this tool to produce a reduced number of false positives. There is still

9 Available at: https://dzone.com/articles/using-machine-learning-for-static-analysis-4, accessed in December 2021.
10 Avaliable at: ttps://snyk.io/news/snyk-announces-acquisition-of-deepcode/, accessed in December 2021.
11 Available at: https://www.grammatech.com/products/source-code-analysis, accessed in December 2021.
12 Available at: https://www.hcltechsw.com/appscan, accessed in December 2021.

potential for development on the integration front, according to critics, as it currently lacks a functional Jenkins plugin.

Kiuwan Code Security

Code Security[13] is a SAST solution from Kiuwan. It supports over 30 programming languages, scans locally and then shares results in the cloud. It is known for its fast scan and results delivery. It can remediate vulnerabilities and the user can customize its environment.

Snyk

Snyk[14]'s main goal is to work for developers and solve the major problems of traditional SAST products: they are too slow, with scans taking several hours, and they have a history of poor accuracy and false positives. Snyk Code employs a semantic analysis AI engine that learns from millions of open-source commits and is combined with Snyk's Security Intelligence database, resulting in a constantly growing code security knowledge base that reduces false positives to near zero [15].

Coverity

Coverity[16] is a SAST solution from Synopsys. It supports 22 languages and more than 70 frameworks. Rapid Scan, a rapid, lightweight static analysis engine, is included in Coverity and may be used to scan online and mobile apps, microservices, and infrastructure-as-code (IaC) settings.

Fortify

Fortify [17] is compatible with 21 programming languages and can detect over 900 different types of vulnerabilities. To automate all validation operations and decrease the danger of false positives, it employs a unique dataflow analysis technique and machine learning algorithms. Fortify double-checks the fix to guarantee that the vulnerability was indeed removed and that no new issues have arisen in its place. It performs this automatic double-checking of all potential fixes using the results of prior audits as well as a comprehensive knowledgebase. The tool translates the format of the source code, scans it, then gives a

13 Available at: https://www.kiuwan.com/code-security-sast/, accessed in December 2021.
14 Available at: https://snyk.io/
15 Available at: https://docs.snyk.io/products/snyk-code
16 Available at: https://snyk.io/, accessed in July 2022.
17 Available at: https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer, accessed in July 2022.

detailed report. The user can't add rules and has to upload codes manually. Fortify can integrate with the Eclipse IDE or Visual Studio as well and has a mechanism for interaction with continuous integration management systems, which allows automatic generation of error reports.

CTool

CTool is a tool that is used to scan C, C++, Java code and bytecode. The tool can be executed from the command line or via a full application. It also provides different interfaces such as an IDE plugin, or a web page. Each vulnerability is reported along with a priority score, its type, the code location, and information on its severity and sometimes fixes are suggested. The GUI cannot be customized, since it is a web page. The tool provides the ability to export a report in xml, html, or pdf format, and to customize reports. It also allows the users to customize the analysis by choosing which checkers to run, and set code annotations that guide the analysis at runtime[Smith et al. (2020)].

Veracode

Veracode[18] The static binary analysis engine models the binary executable into an intermediate representation, which is then verified for security flaws using a set of automated security scans. Their SAST tool provides fast static analysis with automated security feedback, across the development environment (IDE integration) and from the CI/CD pipeline.

Checkmarx

Checkmarx CxSAST [19] support 27 programming languages and frameworks. Some tools that rely on only lexical analysis have the tendency to output many false positives because they do not take into account the semantic. This tool overcomes this lack by using the Abstract Syntax Tree (AST) of the program being evaluated [Baptista et al. (2021)]. Besides finding vulnerabilities, based on those results, it suggests recommendations on how to fix problems with linking to a graphic scheme. It can be integrated with various development environments (Eclipse, IntelliJ, Visual Studio, etc.), build servers (Jenkins, CLI, Bamboo, Maven, TeamCity), version control systems (Bitbucket, etc.), and bug tracking (Atlassian Jira).

18 Avaliable at: ttps://www.veracode.com/, accessed in November 2021.
19 Avaliable at: ttps://checkmarx.com/product/cxsast-source-code-scanning/, accessed in November 2021.

Table 1: Comparative table of SAST tools.

Tool	Licence	Execution	Technologies	Report Format
Flawfinder	Free	CLI	Pattern-matching	HTML/CSV/CLI
NodeJsScan	Free	CLI	Pattern-matching	JSON
Semmle	Free	App/IDE Plugin	Pattern-matching	CSV/SARIF
Brakeman	Free	CLI/ Jenkins Plugin	Unknown	HTML/Generic Issue Import Format
Findbugs	Free	CLI/ Eclipse Plugin	Unknown	xml/html
RIPS	Free	CLI/App	Lexical/Semantic analysis	Unknown
JsHint	Free	CLI/App	Unknown	XML
CodeWarrior	Free	CLI/App	Unknown	Unknown
SonarQube	Free	App	Syntax trees/Semantic API	JSON
DeepCode	Commercial	IDE	Machine learning	Unknown
Reshift Security	Commercial	IDE Plugin/Extension/Repositories	Unknown	Unknown
CodeSonar	Commercial	App	Pattern-matching	SARIF
AppScan	Commercial	App	Machine learning	DB/XML/PDF
Fortify	Commercial	CLI/App/IDE Plugin	Machine learning	XML
CTool	Commercial	CLI/App/IDE Plugin	Unknown	XML/HTML/PDF Format
Veracode	Commercial	App	Unknown	XML
Snyk	Commercial	CLI/App/IDE Plugin	Semantic analysis AI engine	JSON
Coverity	Commercial	IDE Plugin	Unknown	CSV/SARIF/JSON
Kiuwan	Commercial	IDE Plugin/Repositories	Unknown	PDF/CSV
Checkmarx	Commercial	App/IDE Plugin/Others	Lexical/Semantic analysis	JSON/XML/PDF/CSV/RTF

Table 1 summarizes the most relevant characteristics of the SAST tools stated above. Those marked with *unknown* were not found or it's about non revealed information, to the best of our knowledge.

3.1.3 *Types of output*

The most common elements that can be found in a SAST output are:

- The name of the issue;
- Description of the problem;
- CWE related to the issue;
- Location: filename, start and end line, start and end column;
- Query executed that returned this issue;
- Function or type of node where it occurs;
- Severity or impact of the issue.

The majority of the SAST tools reviewed have these elements in their output. However, they can have more elements such as:

- A copy of the source code where the issue is located;
- Suggestions to fix the issue.

Table 2: Comparative table of SAST tools output.

Element	Flawfinder	NodeJsScan	Semmle	Snyk	Checkmarx	A	B	C
Name	✗	✓	✓	✓	✓	✓	✓	✓
Description	✓	✓	✓	✓	✗	✗	✓	✗
CWE	✓	✗	✓	✓	✓	✓	✗	✓
Filename	✓	✓	✓	✓	✓	✓	✓	✓
Start line	✓	✓	✓	✓	✓	✓	✓	✓
End line	✗	✗	✗	✓	✗	✗	✗	✗
Start column	✓	✗	✓	✓	✓	✗	✗	✗
End column	✗	✗	✓	✓	✓	✗	✗	✗
Query	✗	✗	✓	✓	✓	✗	✗	✗
Function or node	✓	✓	✗	✗	✓	✓	✓	✗
Severity	✓	✗	✓	✓	✓	✓	✓	✓
Source code	✓	✓	✗	✗	✗	✗	✓	✓
Suggestion	✗	✗	✓	✓	✗	✗	✓	✗

Table 2 shows the most important elements that are present in the SAST tool outputs. A, B, C denote commercial tools; as their output contain confidential data, their names must be hide.

3.2 TOOLS COMPARISON

Nowadays, there are many choices available in the market so before choosing a SAST tool for a development team it is necessary to take into account some aspects:

- The tool must analyze code in the languages the required applications are written in and must support the framework used by the application so that it integrates easily into the SDLC.
- It is important to check the accuracy of the SAST tools that is intended since it is the weakest point for SAST and there will always be false positives and false negatives.
- A SAST tool should be easily integrated into a CI/CD pipeline or with IDE's and repositories.
- The tool must be customizable. The team must configure the tool to suit their development needs. For instance, they can write new rules to help detect additional security vulnerabilities. They may also need to integrate the SAST tool into their build environment and create dashboards where they can track scan results and generate custom reports.
- A SAST tool that delivers results on larger projects as for smaller ones with the same efficiency.

There are various sites available on the internet which compare SAST tools but not based on all the important criteria referred above. It is a superficial comparison about the advantages and disadvantages of commercial or open source tools, the investment, the speed of scanning and accuracy, they do not report the specific results of the compared tools. Furthermore, they also compare their features, clients reviews and ratings [20], [21], [22].

Another way of testing Application Security Testing tools is by using benchmarks. The OWASP Benchmark Project is a Java test suite designed to verify the speed and accuracy of vulnerability detection tools. It is a fully runnable open source web application that can be analyzed by any type of Application Security Testing (AST) tool, including SAST [23]. After testing SAST tools individually with a benchmark, a comparison between them can be done. There has been some studies that follows this paradigm and reports metrics obtained by the SAST tools comparison. In this specific project, they run the selected SAST tools against the OWASP Top Ten Benchmark designed with the default configuration for each tool, select appropriate metrics to analyze results and finally rank the SAST tools according to the result metrics such as percentage and number of false positives, false negatives and true positives [Higuera et al. (2020)]. There are other approaches using the Juliet Test Suite[24] as it is not only limited to the top 10 vulnerabilities as of the OWASP benchmark [Gentsch (2020)].

A study was conducted aiming at the creation of a platform that combines two known SAST tools in order to improve the performance. The development team had decided to select the combination of SonarQube and SpotBug as the most practical solution with SAST tools. The further development includes SpotBug plugin to a community-version SonarQube and a new SonarQube widget to customize the scanning result. As static analysis uses basically whitebox testing to explore source code, this result shows the potential to improve existing tools, and probably towards a universal security static security ruleset[Nguyen-Duc et al. (2021)].

Many evaluators follow a common method when deciding which static analysis tool will perform best for their group or business. They test each tool on the same code, compare the findings, then choose the tool that reports the most violations out-of-the-box. Other approach is to scan the code taking several samples written in different languages. Then, developers have to look at both false positives and false negatives, knowing where the vulnerabilities are and compare those found by different tools.

The tool-compare repository[25] available on github, compares static code analysis of IaC tools and integrates some of the features this project intend to implement. In the world of

20 Avaliable at: ttps://www.trustradius.com/static-application-security-testing-sast, accessed in November 2021.
21 Avaliable at: ttps://www.capterra.com/static-application-security-testing-(sast)-software/pricing-guide/, accessed in November 2021.
22 Avaliable at: ttps://www.getapp.com/all-software/static-application-security-testing-sast/, accessed in November 2021.
23 Avaliable at: ttps://github.com/OWASP-Benchmark/BenchmarkJava, accessed in November 2021.
24 Avaliable at: https://www.nist.gov/publications/juliet-11-cc-and-java-test-suite.
25 Avaliable at: ttps://github.com/iacsecurity/tool-compare, accessed in November 2021.

infrastructure-as-code security there are several tools for users to choose from. The goal of this repository is to help compare the different options so that users can choose the tool that best fits their needs.

3.3 SUMMARY

This chapter described the research which consisted in learning about the most well-known SAST tools, what they do, their mode of execution, supported languages, the different technologies used and different report formats. It were also discussed the various SAST tools comparisons found in the literature. Moreover, online platforms to compare those tools, as well as the approaches used by developers to do this analysis are presented.

4

PROPOSED APPROACH

This chapter presents a functional requirements elicitation and gives a quick overview of the proposed architecture for the implementation of the system as well as the technologies needed for achieving the proposed goals.

4.1 REQUIREMENTS

A functional requirement is a need that has to be fulfilled in order to deliver an operational capability. Below are the functional requirements for this proposal:

- Upload a configuration file that defines how a new tool is executed;
- Upload a reader that convert a new tool output to a temporary struct;
- Upload a scan output, attached to a new project or an existing one;
- Upload a project source code so the scan results can be linked to the related line;
- Scan a project with configured tools;
- Compare scans based on many criteria and download the results.

4.2 ARCHITECTURE

Figure 4 describes the first system architecture defined to develop this project:

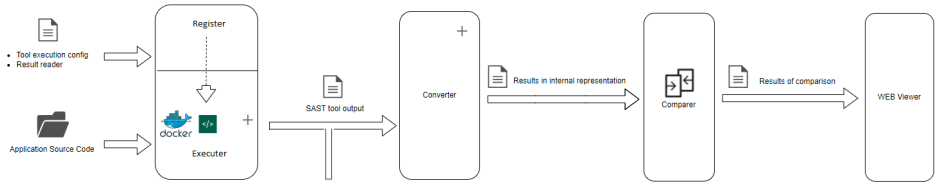


Figure 4: System architeture.

Register - This component will receive and consume configuration files that will instruct how the tools are executed. It will register the tools in the system.

Executer - This component will receive the application source code and analyse it using the tool the user has chosen among the supported tools on the system. The system will execute the scan via docker, command line as well as other options and produce the results.

Converter - This component will receive two types of inputs: the results of the scan using the executer or the output of scanning the same project using a tipically commercial tool, externally. To process the data, the system must be able of reading SAST outputs in various formats such as JSON, HTML, XML, SARIF whether these are customized to the product itself or not. After obtaining the two objects of study, the system will read the data based on result readers provided for each supported tool or generic output format. This will convert the SAST outputs to an intermediate representation so that the system can compare them quickly and efficiently.

Comparer - The system will compare the two objects that contain the vulnerabilities of each scan, indicating what they have in common and highlighting the finding gap.

Viewer - The result of comparison will be presented through a web interface. The results can also be exported as a report on a json format, where the different vulnerabilities present on the two scans are described.

4.3 TECHNOLOGIES

This section introduces the technologies that will be used in this implementation as well as the reasons that led to this choice. The programming language will be Go, the framework Vue for the web interface and Docker to integrate the open-source SAST tools.

4.3.1 *GO*

Go is an open-source programming language designed for building simple, fast, and reliable software, therefore it is the chosen language that will be used for the backend of this project [1] Go was originally designed at Google in 2007. At the time, Google was growing quickly, and code being used to manage their infrastructure was also growing in both size and complexity. Some Google engineers then decided that they needed a new

1 Avaliable at: https://www.golang.org, accessed in December 2021.

programming language focused on simplicity, efficiency in both compilation and execution and effectiveness in writing reliable and robust programs [Alan A. A. Donovan (2015)]. Go is an open-source project so code for its compiler, libraries and tools are avalilable to everyone. It also has an active community that develops modules and libraries and helps newcomers, contributing to the Go project itself.

Here is an example of the Fibonacci Sequence in Go [2]:

```
func FibonacciLoop(n int) int {
    f := make([]int, n+1, n+2)
    if n < 2 {
        f = f[0:2]
    }
    f[0] = 0
    f[1] = 1
    for i := 2; i <= n; i++ {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}
```

4.3.2 *Vue*

Vue.js [3] is an open-source frontend JavaScript framework for building user interfaces and single-page applications. Vue allows to take a web page and split it up into reusable components, each one having its own HTML, CSS and JavaScript needed to render the page. Official libraries and packages provide advanced functionality necessary for complicated applications such as routing, state management, and build tools.

4.3.3 *Docker*

Docker is an open source software platform for creating, deploying, and managing virtualized application containers on a common operating system. A container is a standard unit of software that packages up code and all its dependencies, libraries and configuration files so the program moves from one computer environment to another rapidly and reliably. A Docker image is a lightweight, standalone, executable package of software that includes all the parameters needed to run an application: code, runtime, system tools, system libraries

2 Avaliable at: https://gobyexample.com/, accessed in December 2021.
3 Avaliable at: https://vuejs.org/, accessed in December 2021.

and settings. It executes code inside a container so containers move between Docker environments with the same OS work with no modifications [4].

4.4 SUMMARY

In this chapter, the proposed architecture for this system was presented together with the functional requirements. Moreover, the proposed technologies were described along with the reasons why they were chosen.

4 Avaliable at: https://www.docker.com/, accessed in December 2021.

5

DEVELOPMENT

This chapter describes all the challenges faced and decisions made throughout the development of the comparing system. Each one of the following sections explains how the main required functionalities were implemented: how to add a new tool and a new reader, read a scan and compare results.

5.1 CONFIGURING A NEW TOOL

Besides the existing tools in the system, the user can add a new tool in order to scan projects with it. For this purpose, the system needs the tool name, that must be unique, and information about how it is executed.

This system can receive instructions for two types of tools:

- **The ones that can be executed by command line**
  The user must provide a configuration file with the following information: which commands have to be executed in order to install the tool, the commands to execute the tool with a specific string that will be later replaced for the project path and the output format.

```
1  {
2      "install":["pip install flawfinder"],
3      "scanProject": "flawfinder --sarif {path_to_project}",
4      "format": ".json"
5  }
6
```

- **The ones that can be pulled by a docker image**
  In addition to the case above, the user must provide the name of the docker image to pull and the path to the output.

```
    {
        "image":"imageName",
        "format":".json",
        "scanProject": "{path_to_host_folder_to_scan}:/path tool:latest
    scan -p '/path' -o '/path/'",
        "pathToOutput": "/path/results.json"
    }

```

5.2 READING A SCAN

For the integration of some readers in the system, it was necessary to analyze various SAST outputs and study how the above values could be retrieved.

There are plenty of formats for a SAST scan output and it is common for a tool to deliver many types such as xml, json and csv.

Moreover, some tools present the same type of output, for example SARIF but don't fill all the fields available on this format. A good example of this behaviour is the SARIF output of Flawfinder, Snyk and Semmle:

- Flawfinder and Semmle outputs refers only to the start line, start and end column while Snyk's refers also to the end line.
- Snyk output also presents the code flow that precedes the result since the first node until the last.

Besides, there are several ways of grouping the same type of results. For example, CxSAST groups the same CWE and query results in an array while Flawfinder presents 1 result for each vulnerability found.

This different type of delivering results can be seen next:

Flawfinder - Not grouping

```
{
    "results": [
        {
          "message": {
            "text": "buffer/sprintf:Does not check for buffer overflows (CWE-120)."
          },
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
```

```
                "uri": "test.c",
              },
              "region": {
                "startLine": 21,
                "startColumn": 2,
                "endColumn": 39
              }
            }
          }
        ],
      },
      {
          "message": {
              "text": "buffer/scanf:The scanf() family's %s operation, without a limit specification, permits buffer overflows (CWE-120)."
          },
          "locations": [
              {
                  "physicalLocation": {
                      "artifactLocation": {
                        "uri": "test.c"
                      },
                      "region": {
                        "startLine": 25,
                        "startColumn": 2,
                        "endColumn": 17
                      }
                  }
              }
          ]
      }
  ]
}
```

CxSAST - Grouping by CWE-id and query


```
{
"Queries": [
    {
        "Metadata": {
            "Id": 5587,
            "QueryName": "Buffer_Overflow_Unbounded_Format",
            "GroupName": "CPP:Cx:CPP_Buffer_Overflow:0",
            "Severity": "Critical",
            "CweId": 120
        },
        "Results": [
            {
              "Nodes": [
                {
                  "Column": 25,
                  "FileName": "test.c",
                  "FullName": "bug",
                  "Length": 3,
                  "Line": 20,
                  "MethodLine": 14,
                  "Name": "bug",
                  "NodeId": 129,
                  "DomType": "UnknownReference"
                }]
            },
            {
                "Nodes": [
                    {
                      "Column": 2,
                      "FileName": "test.c",
                      "FullName": "scanf",
                      "Length": 5,
                      "Line": 25,
                      "MethodLine": 14,
                      "Name": "scanf",
                      "NodeId": 184,
                      "DomType": "MethodInvokeExpr"
                    }
                ]
            }
        ]
    },
    {
        "Metadata": {
            "Id": 5556,
            "QueryName": "Buffer_Overflow_Unbounded_Buffer",
            "GroupName": "CPP:Cx:CPP_Buffer_Overflow:0",
            "Severity": "Critical",
            "CweId": 120
        },
        "Results": [
            {
                "Nodes": [
                    {
                    "Column": 7,
                    "FileName": "test.c",
                    "FullName": "f",
                    "Length": 1,
                    "Line": 32,
                    "MethodLine": 14,
                    "Name": "f",
                    "NodeId": 242,
                    "DomType": "UnknownReference"
                    }
                ]
            }
        ]
    }
]}
```

Other aspect that can take many forms in a scan and it is vital to compare results, is the path to the file.

For example, CxSAST output represents the file name in the following way:

"FileName": "\\cinatra-master\\connection.hpp"

This path has double backslashes and does not start with the project name. It starts with the first inner folder. On the other hand, Flawfinder presents the path to the file with a single slash and starting in the outter folder where the project is:

"uri": "Projects/cinatra-master-cpp17/cinatra-master/websocket.hpp"

To fix this incoherence between integrated tools in the system and uploaded scans, it was necessary to normalize the file path in order to be possible to compare results. When converting the output results to an intermediate structure, all backslashes are replaced by a single slash and the string must start in the project folder that is identified by its name. This change covers the majority of cases that came across the development of the system.

Taking into consideration Table 2, the chosen essential elements for a stable comparison of results were the following, since they are the most common between the analyzed outputs:

- The name of the issue
- CWE related to the issue
- Location: start line only
- Filename
- Function or type of node where it occurs

In addition and as a suggestion of an Application Security team, it can be helpful to have the classification of the issue (false negative, false positive, true positive or unclassified) that is added manually and evaluated by this type of teams, so they can have extra information about their results when comparing to others.

The structure that defines a vulnerability for comparison is as shown in the example below:

```
1    {
2        "Issue": "Use of Externally-Controlled Format String",
3        "File": "/program.c",
4        "InitialNode": 9,
```

```
        "TypeNode":"format/printf",
        "CWEList" : ["CWE-134"],
        "Classification" : "True positive"
    }
```

5.3 ADD A NEW READER

In order to interpret an unknown scan output format, the user must upload the results reader. This reader is a *zip* folder that contains a Dockerfile so the program will run under Docker. It must be able to read the scan and create a json file that contains the vulnerabilities. Each existing tool in the system must have a converter attached, since it would be useless scanning a project without knowing how to fetch the important results.

5.4 COMPARING RESULTS

For this project, *CWE-2000: Comprehensive CWE Dictionary*[1] was used since it has the name of every active CWE, its description and relationships, which groups it is inserted in and what views it is relevant for.
For this comparison, it is important to know every CWE-Id and its relationships between child and parent CWE because the purpose is to know if a certain tool is finding an issue that is related to another issue even though they are not exactly the same.

5.4.1 *Types of comparing results*

By CWE-Id

The most reliable way of comparing issues from a SAST output is by testing the equality of CWE-Ids. If both tools find the same CWE in the same line, then it's about the exactly same issue.

By vulnerability name

Not every SAST tool assigns a CWE-Id to an issue. In that case, the comparison can be done between the name of the issue.

1 Avaliable at: https://cwe.mitre.org/data/slices/2000.html, accessed in July 2022.

For example, CxSAST and NodeJSScan probably identify the same issue in line 2 since NodeJSScan assigns a *Password Hardcoded* problem and CxSAST finds this result by running Use_Of_Hardcoded_Password which is assigned to *Use of Hard-coded Password (CWE-259)*. However, neither of this issues names match completely.

```
1 var dummy_info = {
2     "password": "asdfpiuw981"
3 }
```

The chance of matching vulnerabilities names is not high since many SAST tool has its own definition for an issue.

To solve this problem and having more results when comparing by name, there was an attempt of assigning a CWE for the issue name and then compare by it. However, any attempt of assigning a CWE-Id is unsafe. For example, NodeJsScan output does not refer to CWE. Its issues are grouped by categories: Application Related, Cross-site-scripting and Vulnerable Node Module.


```
1  {
2    "sec_issues": {
3      "Application Related": [
4        {
5        "description": "A hardcoded secret was identified.",
6        "filename": "app.js",
7        "line": 44,
8        "lines":"app.use(session({\n secret:'\uoof1asddfilhpaf78h78032h7sg780dsbovncubuyvqy',\n cookie: {\n secure: false,\n
          maxAge: 99999999999\n }",
9        "path": "\\vulnerable-node-master\\app.js",
10       "sha2":"aa7af4f62702b21938ef01f5201cf1a5cbf6664388181aacfebbe88bfe081271",
11       "tag": "node",
12       "title": "Secret Hardcoded"
13       },
14     ],
15     "Cross Site Scripting (XSS)": [
16       {
17       "description": "The EJS template has an unescaped variable. Untrusted user input passed to this variable results in
          Cross Site Scripting (XSS).",
18       "filename": "bought_products.ejs",
19       "line": 26,
20       "lines": "for (var i = 0; i < products.length; i++) {<\ntd > < % -products[i].product_id % > < /td>",
21       "path": "\\vulnerable-node-master\\views\\bought_products.ejs",
22       "sha2": "86bd333ac85d54b1f83b82aac57e52bd90cfb6623f2dc9b72a0795288157cb08",
23       "tag": "xss",
24       "title": "Unescaped variable in EJS template file"
25       },
26     ]
27   }
28 }
```


It is impossible to attribute a CWE-Id for each issue, since these titles are vague:

- Secret Hardcoded can relate to CWE-259 (Use of Hard-coded Password), CWE-798 (Use of Hard-coded Credentials) or even CWE-547 (Use of Hard-coded, Security-relevant Constants).
- Cross Site Scripting is related to dozens of CWE-Ids.

By vulnerability hierarchy

Considering the hierarchy relationships of every CWE-Id, it is possible to indicate if a result is more in-depth than other.
For instance CWE-79 which refers to Cross-site Scripting issue, has one ancestor and many descendants. It is possible to conclude that any result with this CWE, will be very wide.

```
{
"CWE-79": {
  "Name": "Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')",
  "ChildOf": {
   "CWE-74": "Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')"
  },
  "ParentOf": {
   "CWE-692": "Incomplete Denylist to Cross-Site Scripting",
   "CWE-80": "Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)",
   "CWE-81": "Improper Neutralization of Script in an Error Message Web Page",
   "CWE-83": "Improper Neutralization of Script in Attributes in a Web Page",
   "CWE-84": "Improper Neutralization of Encoded URI Schemes in a Web Page",
   "CWE-85": "Doubled Character XSS Manipulations",
   "CWE-86": "Improper Neutralization of Invalid Characters in Identifiers in Web Pages",
   "CWE-87": "Improper Neutralization of Alternate XSS Syntax"
   }
  }
}
```

By node

The last way of comparing results is by analyzing the node in which the issue resides.

Some tools produce results with the type of node and in that case, if the same node type is present in both compared scans, that is a match.

For example, both CxSAST and Flawfinder identify an issue in line 2, on node *sprintf*. CxSAST finds an Unchecked Return Value (CWE-252) and Flawfinder finds a Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') (CWE-120).

```
1 int demo(char *a, char *b) {
2      sprintf(s, "\n");
3 }
4
```

5.5 SUMMARY

This chapter explained the steps that were taken to implement the main functionalities of the SAST tool Comparer.

6

FINAL PRODUCT

This chapter presents the last version of the prototype that implements the architecture proposed in Chapter 4. It was built according to the implementation details explained along the last chapter.

The system integrates already some readers and tools:

- two open source SAST tools - Flawfinder and NodeJSScan- that run by command line.
- some readers for many tools: Flawfinder (json - sarif), NodeJSScan (json), CxSAST (json and xml), Snyk (json), Semmle (cvs and json) and others refered in Table 2.

6.1 USER INTERFACE

This section presents the user interface by using images as well as notes that help better understand the functionality and design.

6.1.1 *Add new tool*

As seen in Figure 5, this view aims to receive the tool configuration file as well the corresponding reader so the results can be interpreted. This page contains some instructions on the configuration file structure (as refered section 5.1), the necessary files inside the reader and the required output structure.

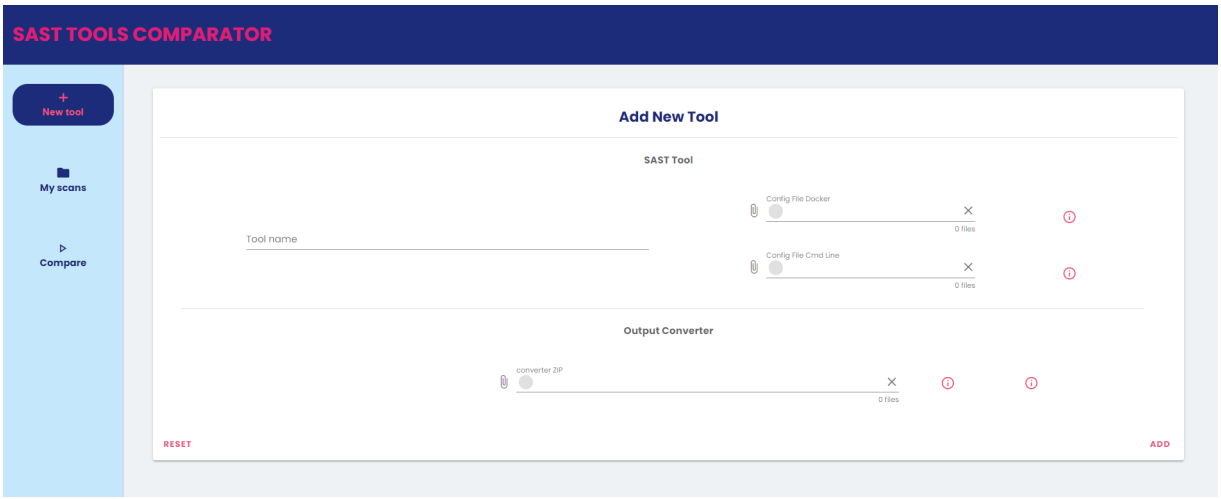


Figure 5: Introduce new tool in the system view.

After uploading these configuration files, the tool is available to scan a project in this system.

6.1.2 *All scans*

This view, Figure 6, shows all scans that were requested in a project scanning or uploaded by the user. Each one has a reference to the tool, the project and the number of results. When opening each panel, it is possible to see each result referring the file name, line, CWE-Id and definition. When clicking the row on the table, it highlights the corresponding line in the source code on the right so it is easy to understand what could possibly cause the issue.

There are also two buttons, one for uploading a scan output and the other for scanning a project.

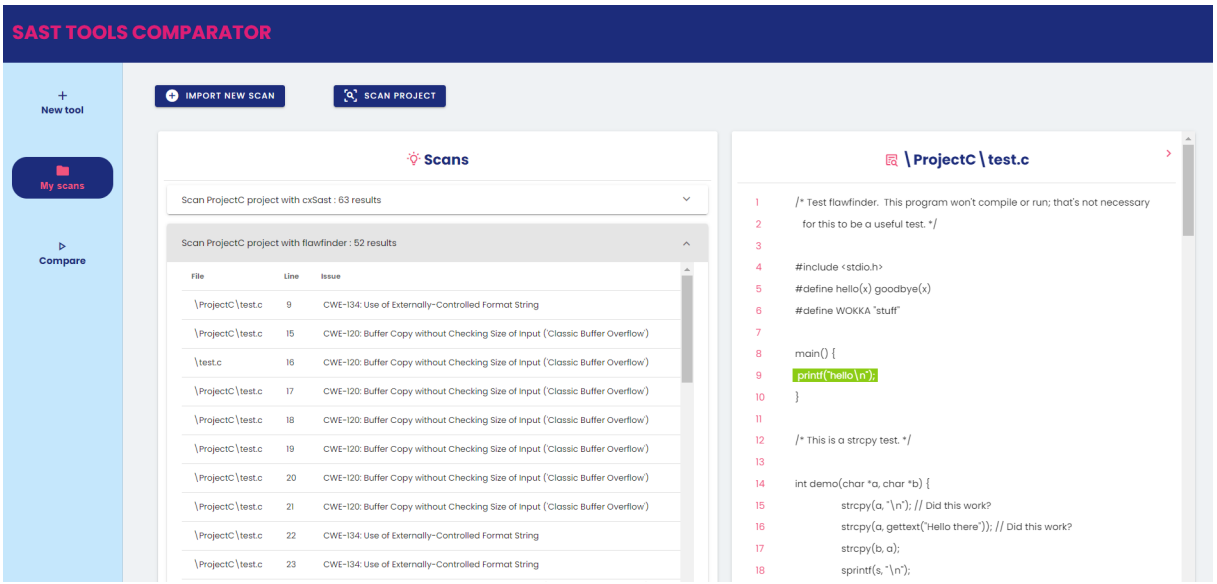


Figure 6: All scans view.

6.1.3 *Upload scan output*

To upload a scan output, it's required to choose a project that is already listed and choose an existing reader to convert. The checkbox must be ticked when the output results uploaded are already in the correct structure to present them, in other words, there is no need to convert the results. These inputs can be seen in Figure 7.

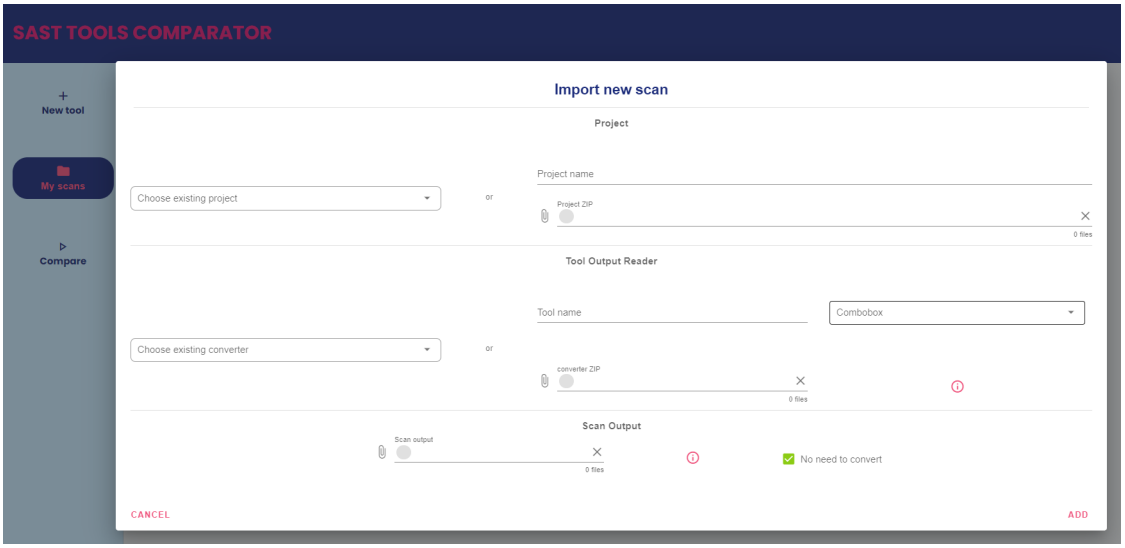


Figure 7: Upload scan output view.

6.1.4 *Scan project*

To scan a project, it's required to choose a project that is already stored locally or upload a new one and choose an existing tool in the system, as seen in Figure 8. At this moment, it is possible to scan a project with Flawfinder and NodeJsScan.

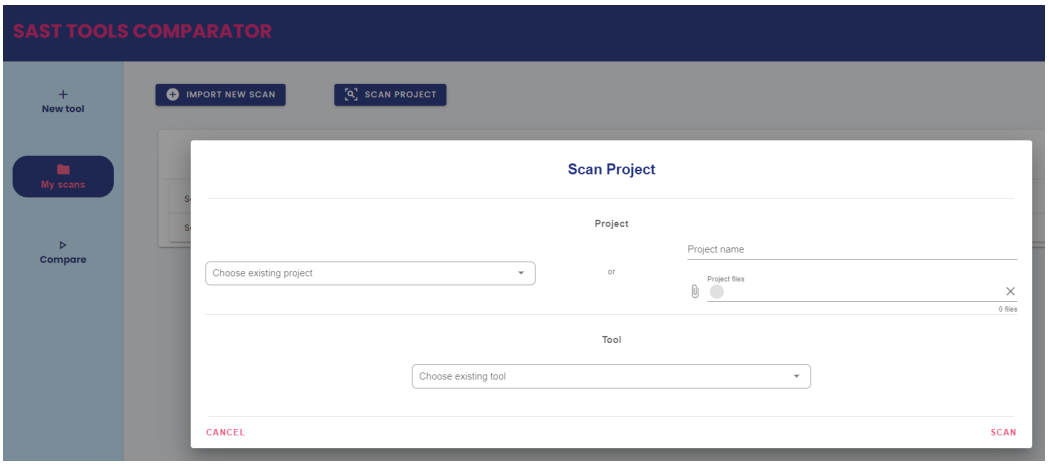


Figure 8: Scan project view.

After having at least 2 scans of the same project, the user can start comparing results.

6.1.5 *Compare results*

The button *New Comparison* shown in Figure 9 opens a dialog in which it's required to choose two scans to compare. These must be scans from the same project, otherwise the comparison would not make sense. The comparison is based on CWE-Ids by default but the user can choose other criteria (hierarchy, name and node) and its order.

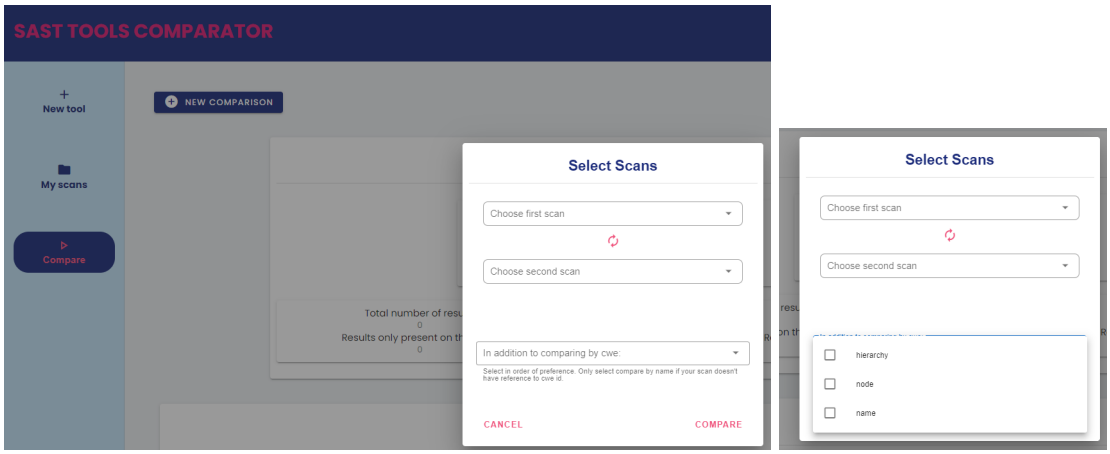


Figure 9: Compare two scan outputs view.

After finishing the comparison, a success message pops up and the results become available.

These results can be downloaded by clicking the button on the right as shown in the image below.

Figure 10 shows some statistics about the comparison:

- Total of different results: sum of the number of results only present in first and second scan.
- Different results ratio: fraction between different and equal results.
- Which tool catches more results and the percentage of this majority.
- Precision: each criteria has a percentage of accuracy. This metric gives a perception of how similar these results are.

*Number of equal results by CWE * 1*
(both tools identify the exact same issue)
+
*Number of equal results by hierachy * 0.5*
(tools identify a related issue)
+
*Number of equal results by name * 0.7*
(tools identify same named issues, the probability of being the exact same are high)
+
*Number of equal results by node * 0.1*
(tools identify a different issue on the same node)

Dividing by total number of results

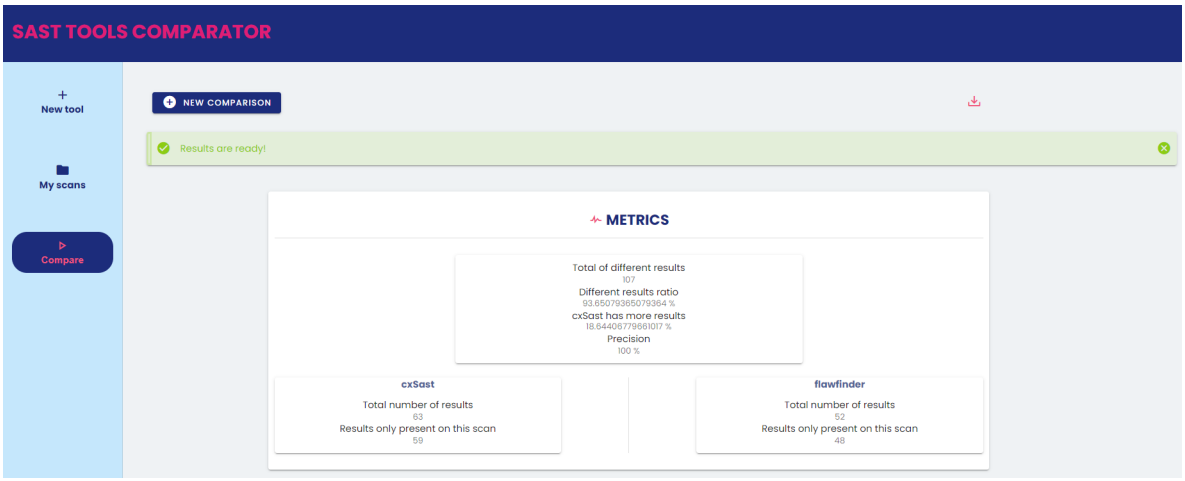


Figure 10: Metrics view.

This view, as seen in Figure 11, shows all the different results present in both scans.

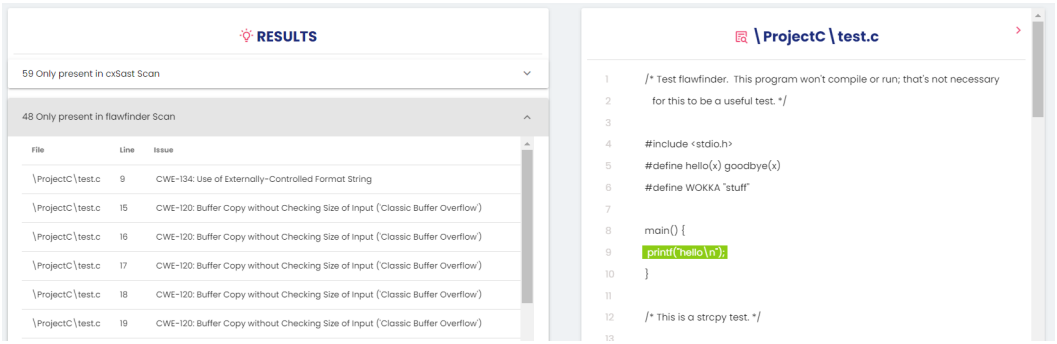


Figure 11: Different results.

This panel as shown in Figure 12 shows all equal results by CWE.

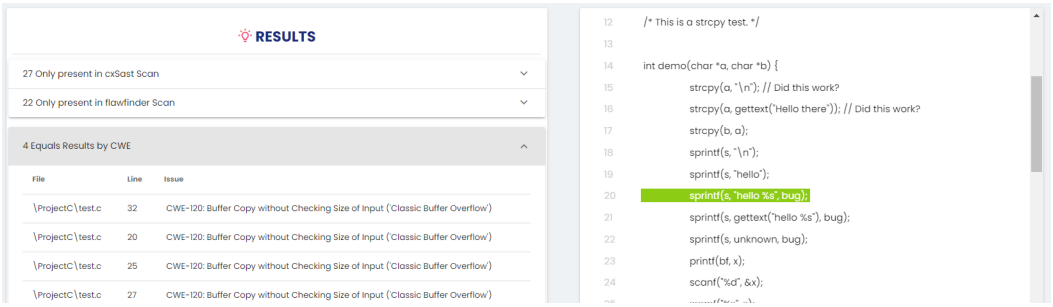


Figure 12: Equal results by CWE panel.

The equal results by hierarchy can be observed in Figure 13 and Figure 14 as the first one represents the parent issue and the second one represents the child issue.

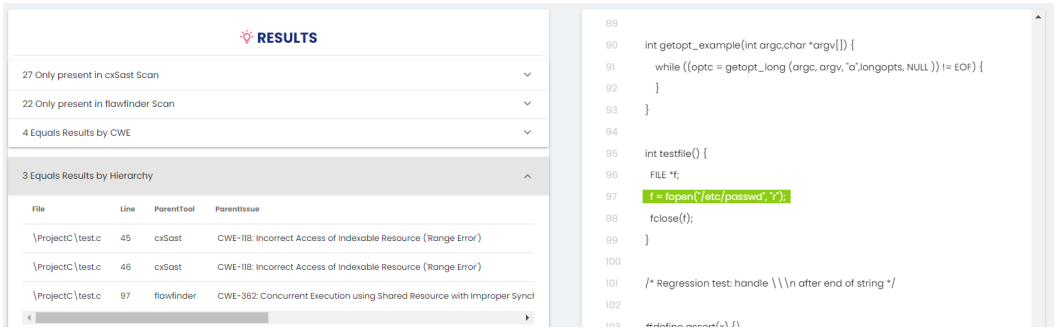


Figure 13: Equal results by Hierarchy - Parent Issue view.

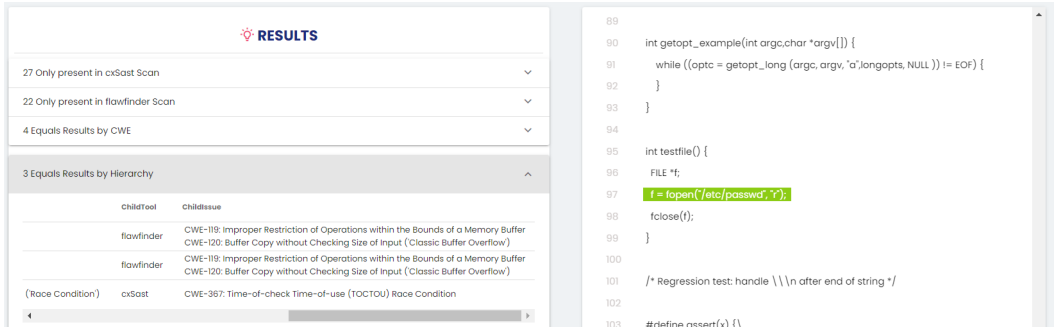


Figure 14: Equal results by Hierarchy - Child Issue view.

Figure 15 and 16 shows the equal results when comparing by node. Figure 15 represents CxSAST issue while figure 16 represents Flawfinder issue for the node *getopt_long*. Each one attributes different CWE-Ids.

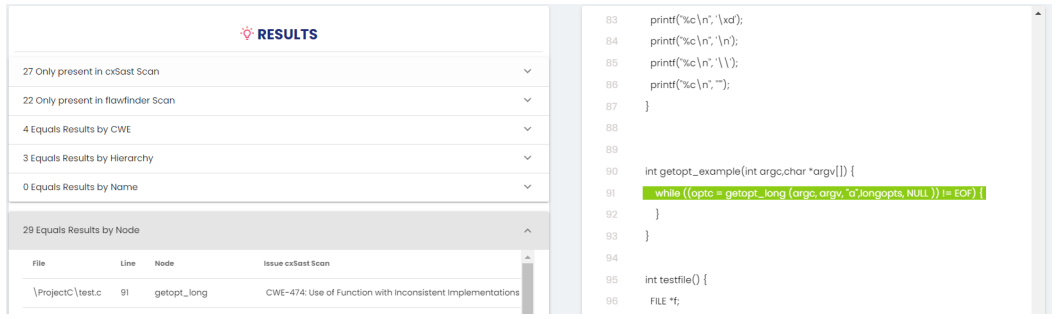


Figure 15: Equal results by Node - CxSAST Issue view.

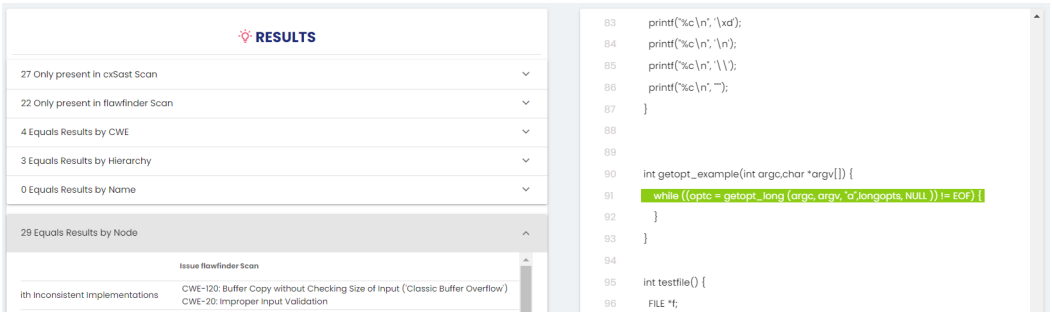


Figure 16: Equal results by Node - Flawfinder Issue view.

6.2 SUMMARY

This chapter showed all the interface views as well as explanations on how to get the results.

7

CASE STUDIES

This chapter is devoted to the validation of the tool developed using for that purpose 3 case studies. Each one compares Checkmarx SAST tool against external tools, Flawfinder, Snyk and Semmle respectively.

7.1 CHECKMARX - FLAWFINDER

To compare results between these two tools, 3 projects were chosen: *cinatra-master-cpp17*[1], *rdkcmf*[2] and *insecure-coding-examples*[3] as seen in figure 17.

Each project was scanned with CxSAST (externally to the comparer tool) and the results were uploaded. Then, the same projects were scanned using Flawfinder, directly from the SAST Tool Comparer.

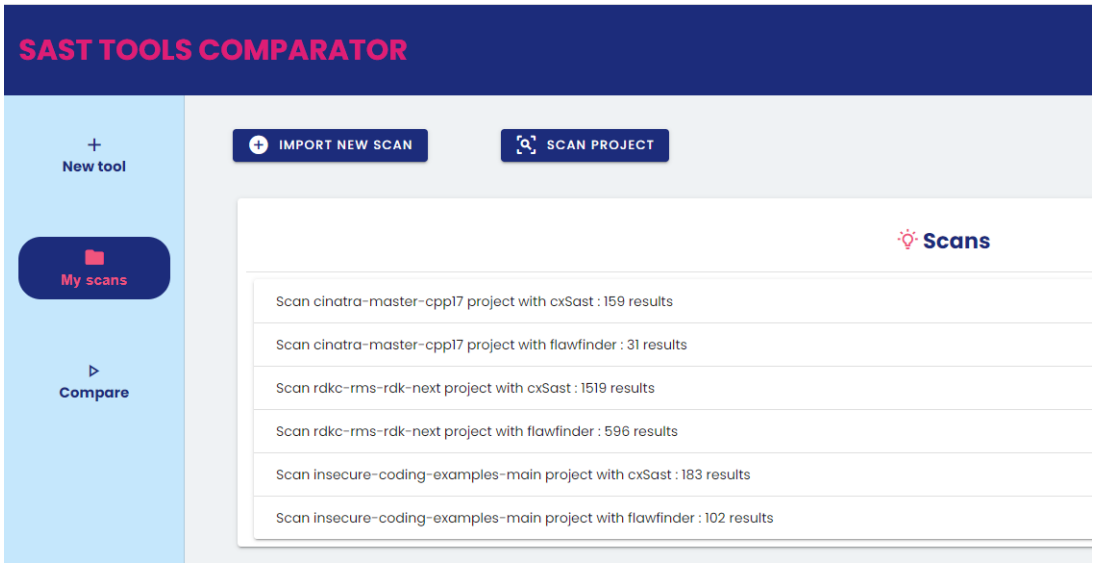


Figure 17: Compare CxSAST and Flawfinder results - scans.

1 Avaliable at: https://github.com/purecpp-org/cinatra, accessed in August 2022.
2 Avaliable at: https://github.com/rdkcmf/rdkc-rms, accessed in August 2022.
3 Avaliable at: https://github.com/patricia-gallardo/insecure-coding-examples, accessed in August 2022.

Figure 18 shows the results of *cinatra-master-cpp17* comparison.

There are 141 different results as 133 are exclusive of CxSAST and 8 from Flawfinder. There were not any results with the same CWE-Id but there were 8 hierarchy matches. In addition, there were 18 nodes where these tools identified different CWE-Ids. These results lead to a precision of 22%.

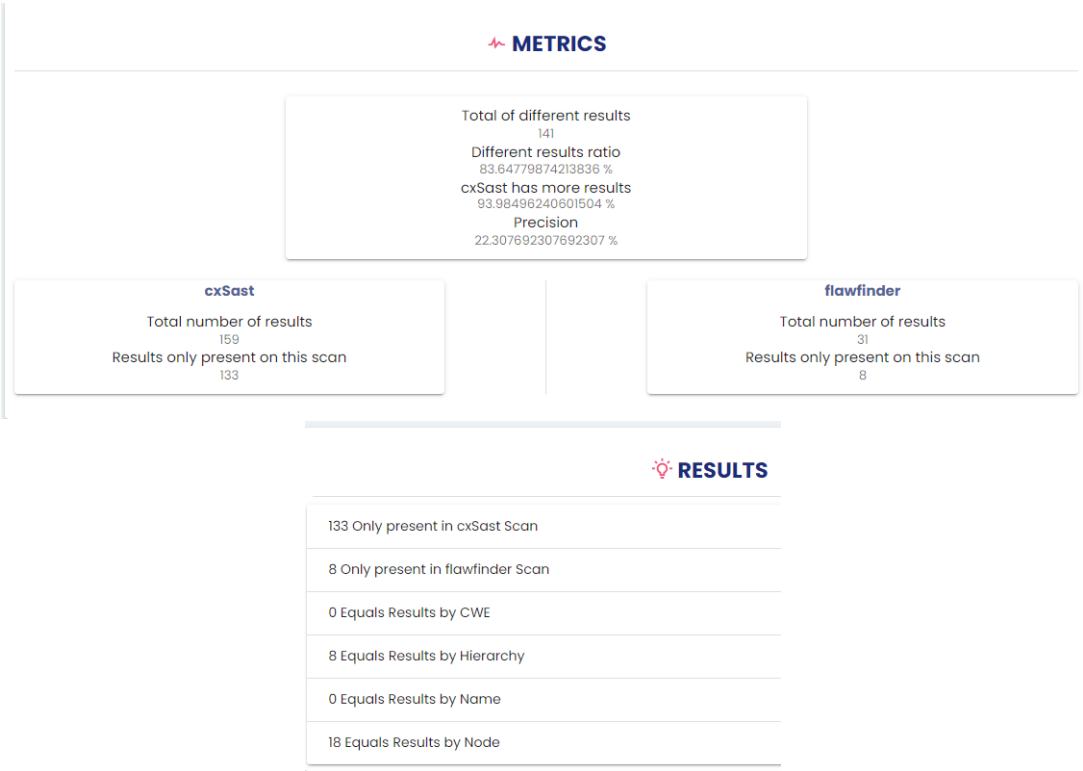


Figure 18: cinatra-master-cpp17 comparison - results.

Figure 19 shows the results of *rdkcmf* comparison.

There are 1836 different results as 1372 are exclusive of CxSAST and 464 from Flawfinder. There were not any results with the same CWE-Id but there were 26 lines where CxSAST and Flawfinder found related issues. Moreover, there were 121 nodes where these tools identified different results and although there are no relationships between these results, they may be good findings on both sides. These results lead to a precision of 17% since there are lots of equal node results.

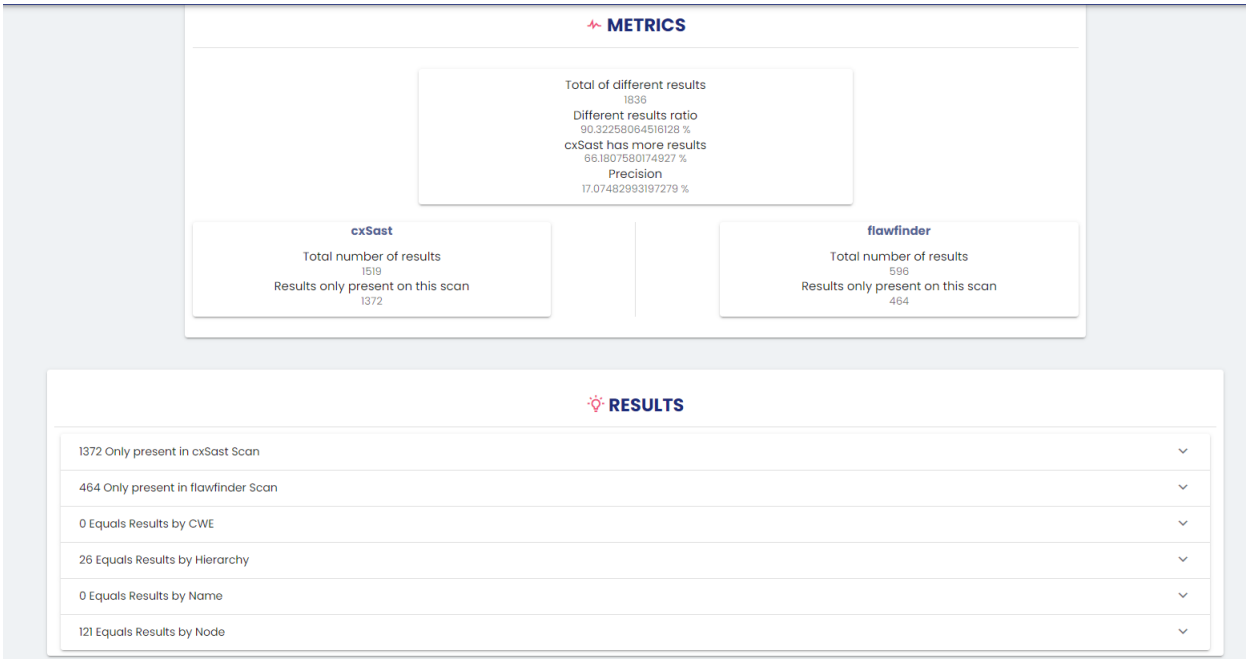


Figure 19: rdkcmf comparison - results.

Figure 20 shows the results of *insecure-coding-examples* comparison.

There are 163 different results as 117 are exclusive of CxSAST and 46 from Flawfinder. There are 11 results with the same CWE-Id and 15 lines where CxSAST and Flawfinder found related issues. Moreover, there were 40 nodes where these tools identified different results. These results lead to a precision of 34% since there are lots of equal node results.

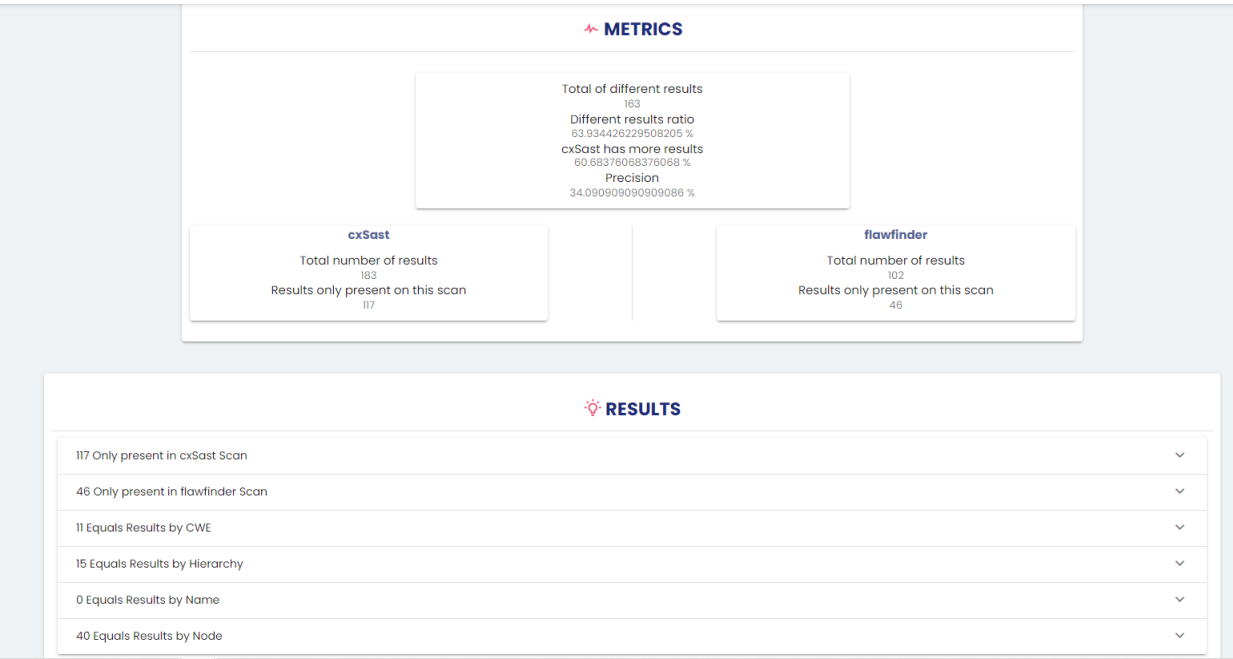


Figure 20: insecure-coding-examples - results.

After analyzing these comparisons statistics as seen in the images above, there are some conclusions that can be taken:

- CxSAST always finds more results than Flawfinder.
- Flawfinder often finds the same CWE-ids. They are CWE-119, CWE-120, CWE-78, CWE-134, CWE-126 among others, which can be a limitation.
- CxSAST and Flawfinder find many different results on the same nodes and that is why the precision of these comparisons are low.
- The equal node results follow the same pattern very often. These pairs of results that appear repeatedly can be worth of investigation:
  - **read** - CxSAST identifies CWE-252 (Unchecked Return Value), Flawfinder identifies CWE-120 (Buffer Copy without Checking Size of Input) and CWE-20 (Improper Input Validation).
  - **printf** - CxSAST identifies CWE-311 (Missing Encryption of Sensitive Data), CWE-497 (Exposure of Sensitive System Information to an Unauthorized Control Sphere) or CWE-256 (Plaintext Storage of a Password) and Flawfinder identifies CWE-134 (Use of Externally-Controlled Format String).
  - **memcpy** - CxSAST identifies CWE-242 (Use of Inherently Dangerous Function), Flawfinder identifies CWE-120 (Buffer Copy without Checking Size of Input).

  - **strlen** - CxSAST identifies CWE-242 (Use of Inherently Dangerous Function), Flawfinder identifies CWE-126 (Buffer Over-read).

- CxSAST and Flawfinder have way more hierarchy results in common rather than CWE equal results. In the majority of these results, CxSAST identifies the parent issue and Flawfinder identifies the child issue. These results are also very repetitive: for example, CxSAST always finds CWE-118 at the same location where Flawfinder finds CWE-119 and CWE-120 and CxSAST always finds CWE-367 where Flawfinder finds CWE-362.

7.2 CHECKMARX - SNYK

To compare results between these two tools, 2 projects were chosen: *WebGoat.NET-Csharp7-master*[4] and *AlegroCart*[5].

Each project was scanned with CxSAST and with Snyk (both externally to the comparer tool) and the results were uploaded as seen in Figure 21.

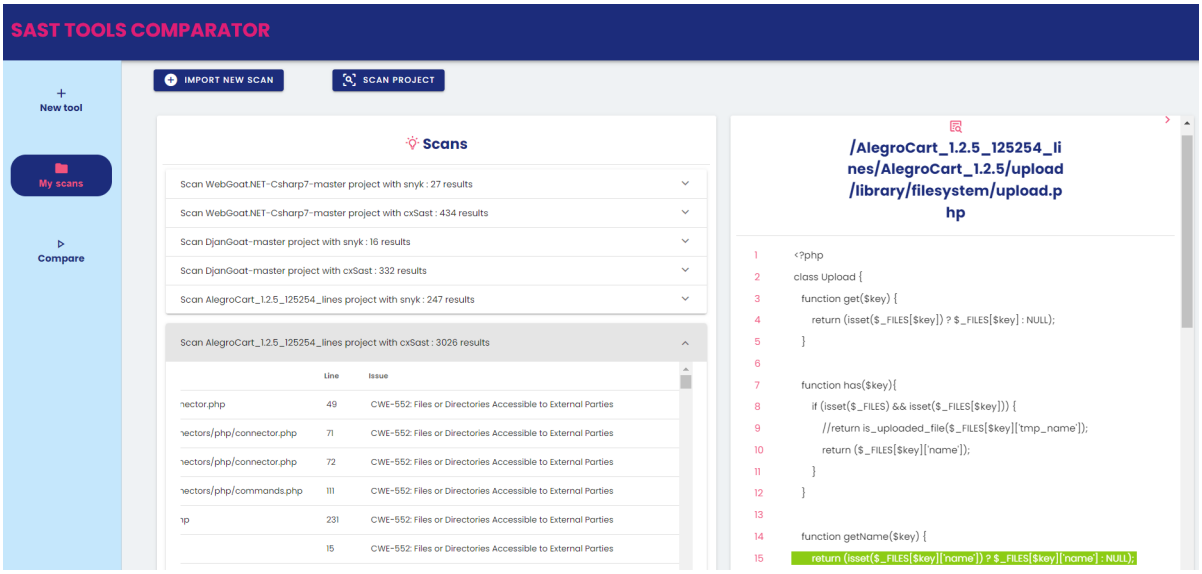


Figure 21: Compare CxSAST and Snyk results - scans.

From this view, it is notable CxSAST finds more results than Snyk.

Figure 22 and 23 show the comparison results between the WebGoat project scans.

4 Avaliable at: https://github.com/jerryhoff/WebGoat.NET, accessed in August 2022.
5 Avaliable at: https://github.com/alegroleo/alegrocart, accessed in August 2022.

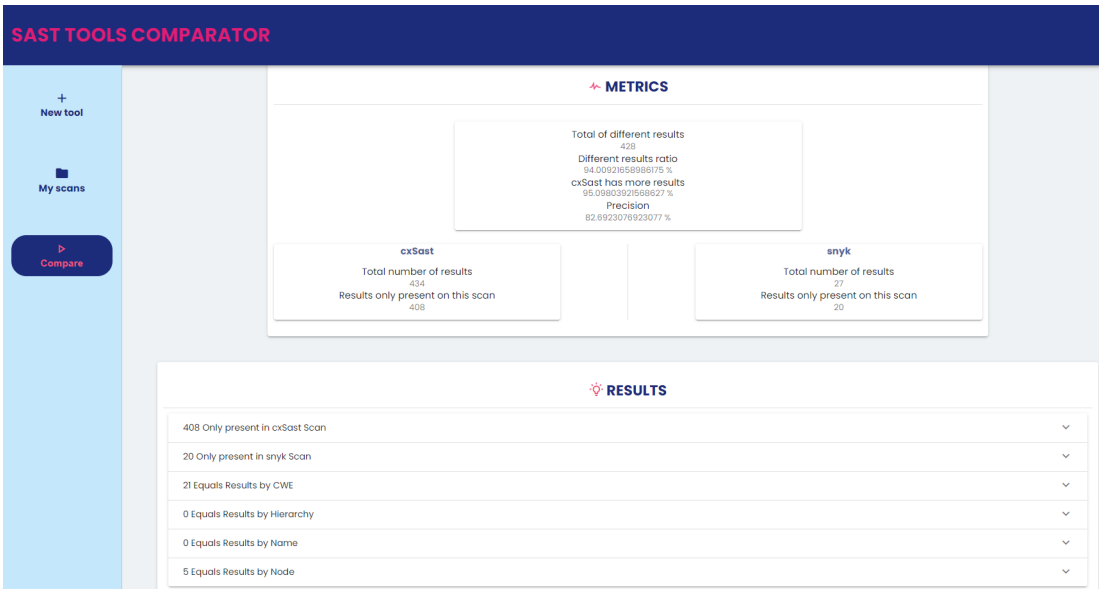


Figure 22: WebGoat.NET-Csharp7-master comparison: CxSAST -> Snyk.

This first comparison (CxSAST vs Snyk) obtained 428 different results as 408 are exclusive of CxSAST and 20 from Snyk. There was a match of 21 results with the same CWE-Id and 5 results in the same node but with different issues associated, which leads to a precision of 83%.

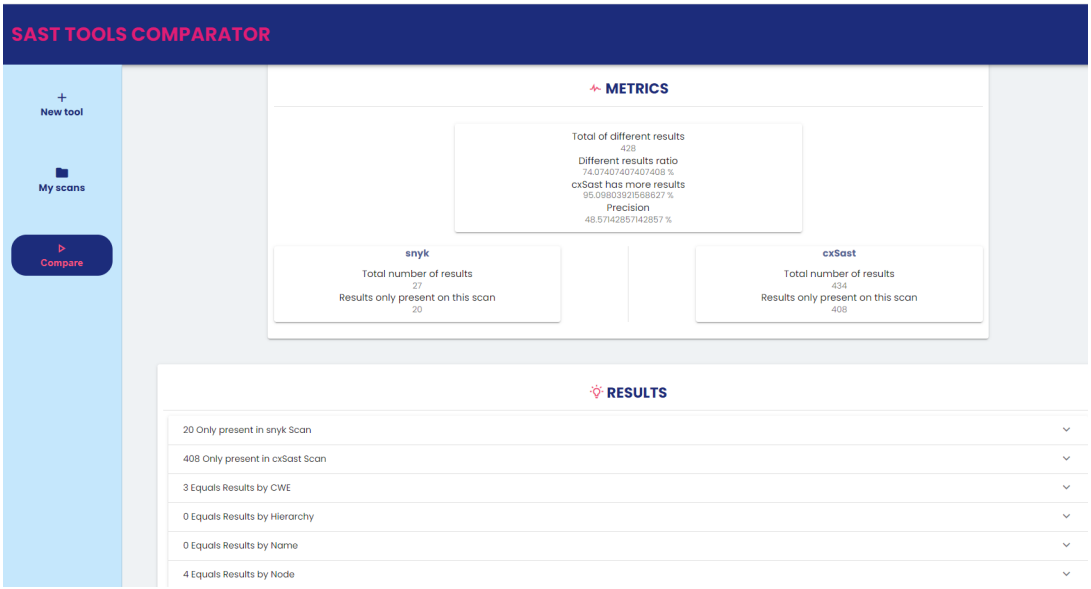


Figure 23: WebGoat.NET-Csharp7-master comparison: Snyk -> CxSAST.

The second comparison (Snyk vs CxSAST) is distinct from the first in that there were only 3 results when comparing by CWE-Id, which leads to a precision of 49% which is lower than

the previous. This result is quite odd since this comparison is based on the same 2 scans, just in different directions.

However, when taking a deep look at these results, it is possible to understand at least one reason for these differences in the numbers: CxSAST finds multiple results in the same row and in different columns, while Snyk displays single results per row.

As the comparison is done only taking into account the line, leaving out the column, in the first case, CxSAST results match Snyk result and join CWE equal results. In the second case, the single Snyk result only matches once with a CxSAST result and despite the comparison being done in two directions (CxSAST results that match Snyk results and vice-versa), this last doesn't add the corresponding results to the list since they would be duplicated. In the second flow, only the different results are added to the list.

Thus, every CxSAST result that exist in the same line but different column, will be discarded and join the *Only present in CxSAST scan* list.

Figure 24 shows same CWE results on the same project line that appear multiple times (line 30 and 41) but are located in different columns, in CxSAST scan, while in Snyk it is a unique result.

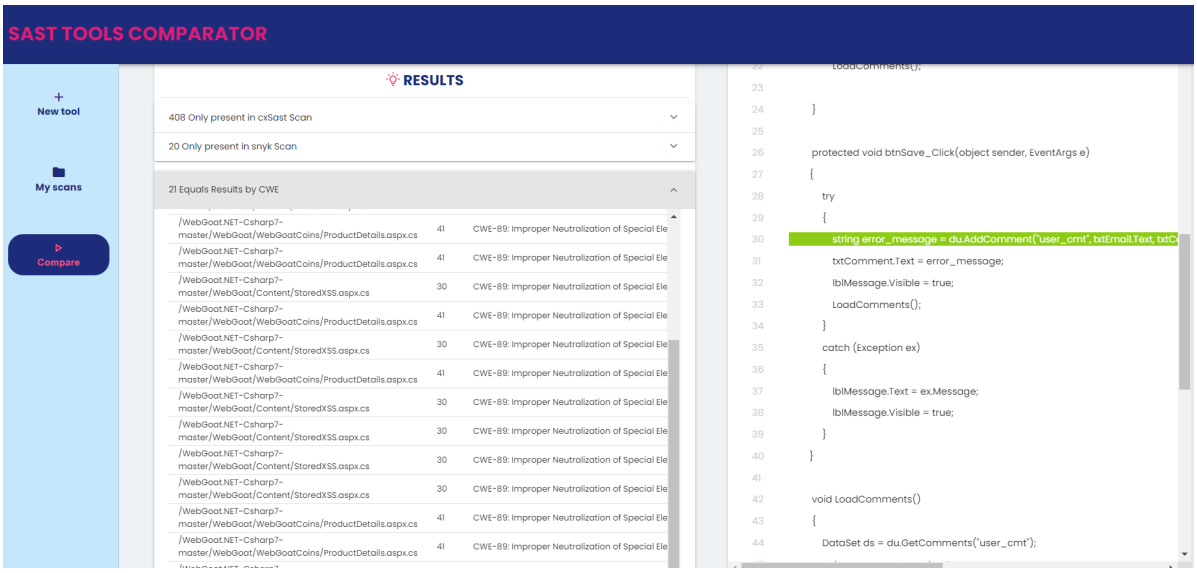


Figure 24: WebGoat.NET-Csharp7-master comparison - equal results by CWE.

This handicap could be fixed by comparing not only by line but also by column.

However, this could lead to another drawback: the number of different results could increase exponentially since a slightly devious column value translates to a mismatch.

In addition, not every tool output makes a reference to the column, therefore this problem continues to exist.

Despite not getting 100% precise results, this way it is possible to analyze 1:N and N:1 results by comparing in different orders.

The next comparison, in Figure 25, uses AlegroCart project and obtained 2799 different results as 2789 are exclusive from CxSAST and 10 exclusive from Snyk, 13 results have the same CWE-Id and 224 results are hierarchy related, which leads to a precision of 53%.

CxSAST continues to find more vulnerabilities than Snyk. Looking to equal hierarchy results, CxSAST always identifies the parent issue and Snyk identifies the child issue which can reflect that Snyk presents more specific results that CxSAST.

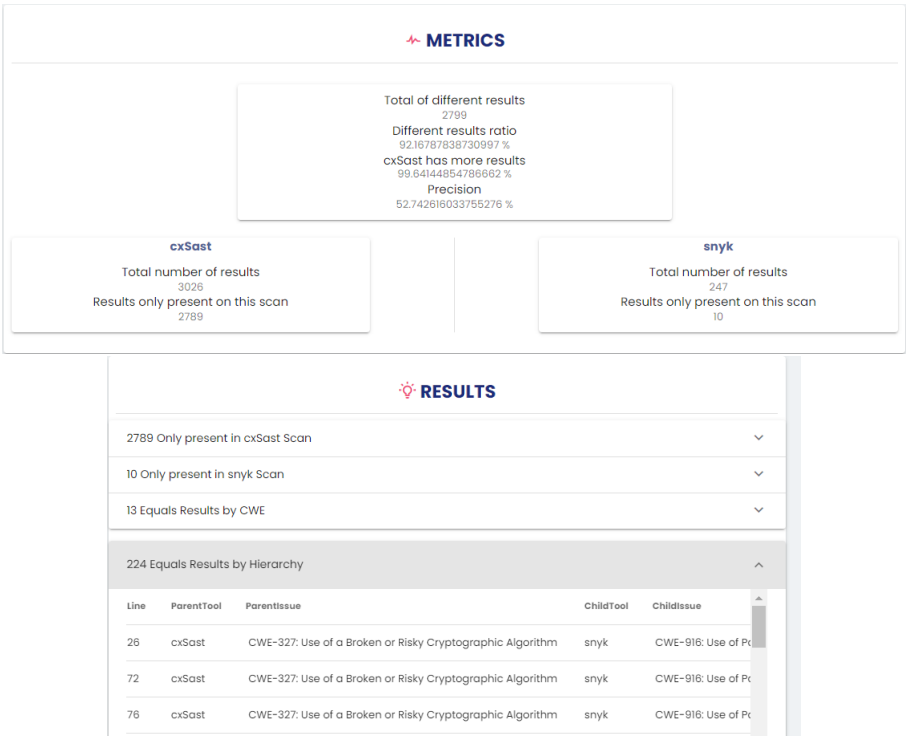


Figure 25: AlegroCart comparison - results.

7.3 CHECKMARX - SEMMLE

To compare results between these two tools, 1 project was chosen: *WebGoat.NET-Csharp7-master*[6].

This project was scanned with CxSAST and with Semmle (both externally to the comparer tool) and the results were uploaded.

6 Avaliable at: https://github.com/jerryhoff/WebGoat.NET, accessed in August 2022.

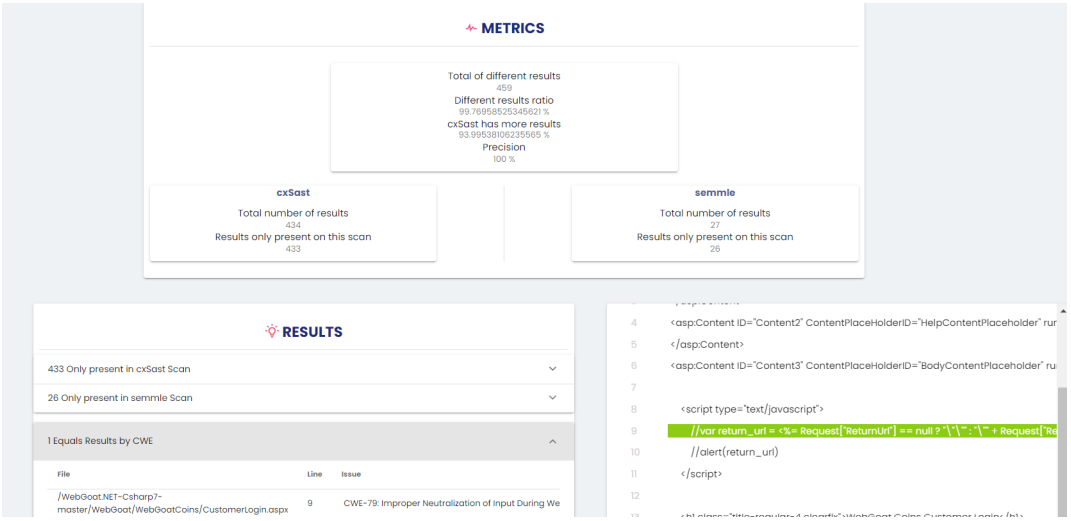


Figure 26: WebGoat.NET-Csharp7-master comparison - results.

Looking at Figure 26, it is noticeable that there are 459 different results, 433 from CxSAST and 26 from Semmle. Comparing by CWE-IDs, only 1 result present in both scans and there aren't results with hierarchy relationships, therefore, these results comparison has an accuracy of 100%.

Taking into account these last comparisons, it is possible to look deeper at the unique results of each tool and see if they are indeed good findings. In case they are, then there is an indication of the existence of false negative results in a tool. Therefore, its rules can be improved to optimize its accuracy or even add new rules to be able to find these results. On the other hand, the same can be done for false positives. If one tool finds a result and another does not, there may exist incorrect or unwanted results. In that case, the rules of that specific tool must be adjusted in order to remove these bad findings.

7.4 SUMMARY

Over this chapter, some comparison results between 4 SAST tools were presented along with an exploratory analysis for each one.

8

CONCLUSION

This chapter is meant to wrap up the master's thesis by providing a summary of the work, the final conclusions and some future work ideas.

The first chapter contains the motivation, objectives, research methodology and presents the document organization of the master thesis.

The second chapter contains some key background concepts that must be understood in order to get along with the work that is going to be presented.

The third chapter consists in investigating about SAST tools, the different options available in the market, what is distinct about each one of them and studying about types of tools comparison done in the past.

The fourth chapter presents and describes the architecture proposal along with its components and functional requirements.

The fifth chapter explains the application development, its main challenges and decisions in order to create a general and extendable system.

The sixth chapter presents the final product and describes each application page, its inputs and outputs, a sort of guide on how the application works.

Finally the seventh chapter presents different case studies that help to understand the distinct type of comparison results, how they could be interpreted and the benefits of such comparison.

Taking into account the research hypothesis, the original premise of building a system that automatically compares SAST tools results by directly scanning with such tools or considering the output findings, was achieved in Chapter 6.

The final product and the case studies analyzed in Chapter 7 were important to demonstrate this application, the type of results that it delivers and what type of meaningful conclusions could be taken.

This project main contribution is to help Cybersecurity companies to evaluate their products considering inputs from other similar tools in the market. What would usually be a manual task, can with SAST tool comparer be made in a semi-automated fashion.

8.1 FUTURE WORK

Throughout the development of this Master's Thesis project, some interesting ideas came up but they were not carried out due to the context and the available time.

However, it would be good to improve this platform. These are some proposals:

- Improve the interface in order to be more efficient and safe and add more functionalities such as remove scans, order results by line, file or cwe.
- The results comparison could be even deeper, for example, to give the option of comparing by column and analyse if results are members of the same view could be another criteria to test the results match.
- This project could be open-source so it would receive valuable inputs from users such as more readers and tools to integrate the system.
- Divide the multiple components of this application so they can be used independently: for example, it could be useful to have the comparison module available outside the application, for possible integration in other tools.
- Provide the possibility of adding a custom vulnerability mapping between two specific tools, for a comparison more adjusted to the user objectives.

BIBLIOGRAPHY

Brian W. Kernighan Alan A. A. Donovan. *The Go Programming Language - Alan A. A. Donovan, Brian W. Kernighan*. 2015.

Bushra Aloraini, Meiyappan Nagappan, Daniel M German, Shinpei Hayashi, and Yoshiki Higo. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software*, 158:110427, 2019.

Murat Aydos, Çiğdem Aldan, Evren Coşkun, and Alperen Soydan. Security testing of web applications: A systematic mapping of the literature. *Journal of King Saud University - Computer and Information Sciences*, 2021.

Tiago Baptista, Nuno Oliveira, and Pedro Rangel Henriques. Using Machine Learning for Vulnerability Detection and Classification. volume 94, 2021.

Samuel Gonçalves Ferreira. Vulnerabilities fast scan, tackling sast performance issues with machine learning. 2019.

Christoph Gentsch. Evaluation of open source static analysis security testing (sast) tools for c, 2020.

Juan R.Bermejo Higuera, Javier Bermejo Higuera, Juan A.Sicilia Montalvo, Javier Cubo Villalba, and Juan José Nombela Pérez. Benchmarking approach to compare web applications static analysis tools detecting owasp top ten security vulnerabilities. *Computers, Materials and Continua*, 64:1555–1577, 6 2020.

Tiago Gomes Rodrigo Moreira Ivo Gomes, Pedro Morgado. An overview on the static code analysis approach in software development, 2009.

John Peyton Kristofer A Duer Jinqiu Yang, Lin Tan. Towards better utilizing static application security testing. 2019.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? 2013.

Jinfeng Li. Vulnerabilities mapping based on owasp-sans: A survey for static application security testing (sast). *Annals of Emerging Technologies in Computing*, 4:1–8, 7 2020.

Christopher J. Fox Michael S. Ware. Securing java code: heuristics and an evaluation of static analysis tools. 2008.

Anh Nguyen-Duc, Manh-Viet Do, Quan Luong-Hong, and Kiem Nguyen-Khac. On the combination of static analysis for software security assessment-a case study of an open-source e-government project. 2021.

Abid Jamil Hamza Tauseef Sahar Ajmal Rimsha Asif Bisma Rehman Sumaira Mustafa Pariwish Touseef, Khubaib Amjad Alam. Analysis of automated web application security vulnerabilities testing | proceedings of the 3rd international conference on future networks and distributed systems. 2019.

Pedro Miguel Lopes Pereira. Automatic fix of source code vulnerabilities. 2019.

F. Guerouate S. El Idrissi, N. Berbiche and M. Sbihi. Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. 2017.

Justin Smith, Lisa Nguyen Quang, Do Google, and Emerson Murphy-Hill Google. *Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security*. 2020.