Semgrep*: Improving the Limited Performance of Static Application Security Testing (SAST) Tools

Anonymous Author(s)

ABSTRACT

Vulnerabilities in code should be detected and patched quickly to reduce the time in which they can be exploited. There are many automated approaches to assist developers in detecting vulnerabilities, most notably Static Application Security Testing (SAST) tools. However, no single tool detects all vulnerabilities and so relying on any one tool may leave vulnerabilities dormant in code. In this study, we use a manually curated dataset to evaluate four SAST tools on production code with known vulnerabilities. Our results show that the vulnerability detection rates of individual tools range from 11.2% to 26.5%, but combining these four tools can detect 38.8% of vulnerabilities. We investigate why SAST tools are unable to detect 61.2% of vulnerabilities and identify missing vulnerable code patterns from tool rule sets. Based on our findings, we create new rules for Semgrep, a popular configurable SAST tool. Our newly configured Semgrep tool detects 44.7% of vulnerabilities, more than using a combination of tools, and a 181% improvement in Semgrep's detection rate.

ACM Reference Format:

Anonymous Author(s). 2024. Semgrep*: Improving the Limited Performance of Static Application Security Testing (SAST) Tools. In *Proceedings of 46th International Conference on Program Comprehension (ICPC 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

It is crucial to detect and patch vulnerabilities in code quickly to reduce the time in which vulnerabilities can be exploited. Vulnerabilities can be difficult to detect and can lay dormant in code for longer than other defects [24]. Static Application Security Testing (SAST) tools are widely used by developers to help detect vulnerabilities [14, 27]. SAST tools are based on static code analysis and are typically fast, resource inexpensive, and can be plugged into continuous integration pipelines, making them popular with developers.

There are limitations associated with using SAST tools, including detecting only a subset of weakness types [18, 21, 22]. As a result, an over-reliance on SAST tools can lead to a false sense of security. This paper aims to understand the limitations of SAST tools by evaluating them on production code with known vulnerabilities. The Static Analysis Tool Expositions (SATE)[1] [6] reports have encouraged the evaluation of SAST tools on production code for many years [17]. Despite this, very few studies have evaluated SAST tools on production code. We respond to this gap by evaluating SAST tools on a real-world dataset of vulnerabilities derived from the National Vulnerability Database (NVD).

Previous studies that evaluate SAST tools have typically used synthetically created benchmarks, rather than real world, vulnerable code. There are some advantages to using synthetic datasets, such as good ground truth and statistically significant results (as a result of high numbers of examples of vulnerabilities). Synthetic datasets are also a solution to the challenge of obtaining a large enough number of real-world vulnerable code examples. However, the results from studies based on synthetic data are not likely to be representative of performance when used in production settings [16]. Our study uses a real-world dataset of vulnerabilities to provide insight into the effectiveness of SAST tools in a production setting.

Individual SAST tools are typically limited by detecting only a subset of vulnerabilities [18, 21, 22]. Combining the use of multiple SAST tools is likely to detect the highest number of vulnerabilities. However, many OSS projects use only a single static analysis tool [14]. Johnson et al. [20] report tool overload as a barrier to using static analysis tools, which may be exacerbated by combining tools.

Li et al. [21] performed an evaluation of seven SAST tools and found over 76.9% of false negatives were attributed to insufficient or missing rules. Therefore, we aim to improve the effectiveness of SAST tools by configuring a single SAST tool to deliver better detection rates than a set of four single tools with standard configuration. To achieve this aim we address the following research questions:

RQ1: What is the detection rate of SAST tools?
We run four SAST tools —GitHub's CodeQL, Snyk, Semgrep and FindSecBugs — on production code with known vulnerabilities. We report each tool's detection rate overall and detection rate per Common Weakness Enumeration (CWE) type to understand which tools are better suited towards certain weaknesses. Unlike previous work, we evaluate SAST tools on production code rather than synthetic vulnerability data.

RQ2: What tool(s) detect the most vulnerabilities?
We determine the detection performance of combining four SAST tools. We identify the distinctiveness of each tool in the set of vulnerabilities it detects, as well as the overlap in vulnerabilities detected across tools. We report the combination of tools which detect the highest number of vulnerabilities.

ICPC 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] SATE is a study of static analysis tool effectiveness aimed at improving tools and increasing public awareness and adoption

RQ3: What is the reason a SAST tool could not detect a vulnerability?
We manually inspect the rule set of the SAST tools to identify why there were detection problems. In particular, we look at the source code associated with the vulnerabilities missed by each tool. We identify detection problems with each evaluated tool and uncover the limitations each has in relation to particular vulnerability types (based on CWE classification).

RQ4: Can a single SAST tool be configured to match or outperform a suite of SAST tools
We use the detection problems identified in RQ3 to configure one tool to detect the vulnerabilities previously detected across our set of four SAST tools. We chose to configure Semgrep as it is the most popular[2], configurable tool. Our contribution is an extended rule set for Semgrep.

Our performance analysis of SAST tools is important for SAST users to understand the limitations of the tool they use. They also need to be aware of the performance improvements available by configuring their tools. Our findings highlight the limitations of four SAST tools, as well as suggested improvements in these tools. Our detection results for each tool, combined results, suggested improvements, and yaml files for an improved Semgrep configuration are available in our replication package[3].

2 BACKGROUND

In this section, we provide background information regarding SAST tools, defining vulnerabilities and their classification, and existing vulnerability datasets.

```
rules:
    - id: use-snakeyaml-constructor
    languages:
        - java
    metadata:
        cwe:
            - "CWE-502: Deserialization of Untrusted Data"
    likelihood: LOW
    impact: HIGH
    confidence: LOW
    message: Used SnakeYAML org.yaml.snakeyaml.Yaml()
         constructor with no arguments, which is vulnerable
         to deserialization attacks. Use the one-argument
         Yaml(...) constructor instead, with
         SafeConstructor or a custom Constructor as the
         argument.
    patterns:
        - pattern: |
            $Y = new org.yaml.snakeyaml.Yaml();
            ...
            $Y.load(...);
    severity: WARNING
```

Listing 1: Semgrep Example

[2]according to GitHub stars, Table 2
[3]https://github.com/lusastevaluation/sastevaluation

2.1 Static Application Security Testing Tools

Static analysis is a method of debugging without executing source code. SAST tools are static analysis tools focused on security testing, i.e. detecting vulnerabilities. SAST tools scan source or compiled code for likely vulnerable code using an established set of rules. Listing 1 presents a Semgrep rule regarding the deserialization of untrusted data using the SnakeYAML library. SAST rules typically contain a name or identifier, an associated weakness type, a set of vulnerable code patterns, and other useful metadata. SAST tools are widely used in OSS, and can be used as Command Line Interfaces, Integrated Development Environment plugins, or plugged into Continuous Integration pipelines.

2.2 Vulnerabilities

The Common Vulnerabilities and Exposures (CVE) Program defines a vulnerability as a flaw in software resulting from a weakness that can be exploited, causing a negative impact on the confidentiality, integrity, or availability of a system [2]. The CVE is a collection of known vulnerabilities catalogued with a CVE Identification number (CVE-ID). The NVD is automatically linked with and augments the CVE, adding useful metadata, such as an associated Common Weakness Enumeration (CWE) type. The CWE is a community-developed list of weakness types and serves as a baseline for vulnerability identification and prevention. Some notable examples include CVE-2021-44228 [4] also known as Log4shell, the vulnerability affecting the Log4J logging tool, and CVE-2017-0144 [5] also known as EternalBlue, an exploit responsible for the WannaCry ransomware attacks.

Vulnerabilities are a particularly dangerous subset of defects, in that their exploitation can cause serious harm to a system and disruption to its users. Munaiah et al. [25] performed an in-depth analysis of defective and vulnerable files and their results suggest defects and vulnerabilities are empirically dissimilar, meaning specialised tools and approaches are required to detect vulnerabilities.

2.3 Vulnerability Datasets

Vulnerabilities are a subset reported to make up between 0.66% and 5% of defects [7, 10, 23]. Developers are often hesitant to disclose their security vulnerabilities on public bug tracking systems, resulting in difficulty obtaining vulnerable code examples. There are three main sources of vulnerable code examples: Synthetic datasets, OSS, and production code with known vulnerabilities.

Synthetically created datasets are a collection of synthetic programs with known flaws. Some examples include: The Juliet Java Test Suite [3], containing 28,881 test cases spread across 112 CWE types (version 1.3) and OWASP NodeGoat [4], which contains OWASP Top 10 security risks affecting node.js. OSS offers vast amounts of source code and contains many vulnerabilities. However, it lacks any ground truth regarding vulnerability locations and types. Production code with known vulnerabilities is contained in datasets such as the NVD, the largest publicly available dataset, containing over 220,000 CVEs with useful metadata such as Common Vulnerability Scoring System (CVSS) scores, CWE types, and

[4]https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[5]https://nvd.nist.gov/vuln/detail/CVE-2017-0144

affected software.

We discuss the strengths and weaknesses of evaluating tools on each source of data in our Methodology Section.

3 METHODOLOGY

In this Section, we describe our methodology for data acquisition, tool selection, the means of verifying whether a tool correctly detected a vulnerability, identifying missing patterns from a tool's ruleset, and configuring and improving Semgrep.

3.1 Data Acquisition

In our study, we use the System Analysis Program Development (SAP) dataset [26] to evaluate SAST tools on production code with known vulnerabilities. We decided to evaluate SAST tools on production code for the following reasons.

Evaluating tools on synthetic datasets has many benefits. Firstly, there is a high confidence that no other errors exist in the code, meaning that any additional vulnerabilities detected can be considered as false positives rather than unknown, true positives. Secondly, synthetic datasets contain very good ground truth. Typically, the vulnerable lines with a corresponding weakness type are known, which enables the automatic verification of results. Thirdly, there are enough examples of vulnerabilities to make the results statistically significant. The benefits listed above allow the calculation of all applicable metrics: precision, recall, discrimination and coverage [17]. While this makes synthetic datasets tempting for evaluators, results from synthetic data are not likely to be representative of performance when used in production settings [16]. Developers have been reported to be sceptical about results from tool evaluations on synthetic benchmarks, expressing the view that synthetic data is too "basic" for real vulnerabilities [9].

Mining OSS is another option to obtain vulnerabilities. However, OSS lacks any ground truth regarding vulnerability locations and types. Since the vulnerabilities are unknown, evaluating SAST tools on OSS prevents knowing the recall. Calculating the precision of the tool becomes a significant manual effort to determine true and false positives. In addition to the considerable time and effort required, labelling security alerts as false positives requires expertise. Baca et al., [13] found that the average developer was unable to correctly identify a vulnerability from the corresponding SAST tool's security warning and required prior experience with that vulnerability.

The SATE V Report [17] identifies the ideal dataset for evaluating SAST tools as representative of production code, containing large amounts of data, and providing good ground truth. However, no such dataset exists. Synthetically created test data lacks realism, while mining OSS is without any ground truth; moreover, there are too few examples of production code with known vulnerabilities to be statistically significant. We aim to identify flaws with current SAST tools to improve their configuration. To simulate these tools in production settings, it is crucial that our data is representative of real vulnerabilities. It would take a considerable amount of time to discern true positives from the large number of warnings from multiple tools. Therefore, evaluating tools on production code with known vulnerabilities is the best option for our study.

CWE-ID	CWE Name	Count
CWE-611	Improper Restriction of XML External Entity Reference	27
CWE-502	Deserialization of Untrusted Data	18
CWE-22	Path Traversal	15
CWE-20	Improper Input Validation	15
CWE-79	Cross-site Scripting	14
CWE-94	Improper Control of Generation of Code	10

Table 1: The top five most frequent weakness types in our dataset.

Ponta et al. [26] manually curated the SAP dataset, containing 624 vulnerabilities affecting Java products. The SAP dataset contains a vulnerability identifier, repository URL, and commit identifier. We add a corresponding CWE-ID to each vulnerability by scraping the NVD website. Thereby, we excluded 29 vulnerabilities that were not submitted to the CVE and NVD. For each of the remaining 595 vulnerabilities, we attempted to clone the repository, but excluded 47 since their repository were no longer available. Using the vulnerability fixing commit identifier, we checkout the parent commit, giving us a version of a repository with a known vulnerability. We exclude 46 vulnerabilities due to the commit no longer belonging to any branch of the repository, which prevents checking out of that commit. GitHub's CodeQL and FindSecBugs both require a fresh build every time a scan is performed. Therefore, we removed 332 vulnerabilities that could no longer build. The data after these exclusions contained 170 versions of repositories with a known vulnerability, spread across 46 projects, 87 repositories, and 36 weakness types. Table 1 lists the five most frequent weakness types in our data (The full breakdown of CWE types can be found in our replication package [6]). These weakness types are the most dangerous vulnerabilities affecting the Java programming language, regularly appearing in the CWE's Top 25 Most Dangerous Software Weaknesses lists [1].

3.2 Tool Selection

To discover the latest SAST tools applicable for our evaluation, we started with the list produced by OWASP [5] comprising 114 available tools.

- We considered the 65 versions of free or open source tools since commercial tools typically have license restrictions for reporting results and may hide their rules hindering our progression of RQ3.
- We excluded any tool not developed for analysing Java files, leaving 31 versions of tools remaining.
- We removed 8 duplicate versions of the same tool, for example, an IDE plugin and CLI version.
- To obtain results of current SAST tools, we removed 8 tools that were no longer available or actively developed (no new release in the past two years).
- We excluded five tools — PMD, SpotBugs, Insider, MobSF and Graudit — that were not focused on security (PMD, SpotBugs, Graudit, MobSF), or did not support CWE classification of vulnerabilities (Insider uses OWASP top 10).

[6] https://github.com/lusastevaluation/sastevaluation

Tool	Watch	Fork	Star
Semgrep	90	520	8800
CodeQL	217	1500	6400
Snyk Code	166	577	4600
FindSecBugs	90	464	2100
Horusec	49	156	948
HuskyCI	30	128	536
BetterScan	15	17	525
HCL AppScan	7	6	9
Fluid's Attack Scanner	-	-	-

Table 2: SAST tool GitHub usage.

- We excluded SonarQube from our analysis as it requires Java version 17 and we had compatability issues with building many projects.
- Table 2 lists the remaining nine tools with their GitHub tracking statistics. We focus on tools developers are using, so we limit our study to the tools with over 1,000 stars on GitHub. Leaving: Semgrep, CodeQL, Snyk Code, and FindSecBugs.

3.3 Tool Verification

We scanned all 170 repositories containing a known vulnerability with each tool using their standard configuration. For CodeQL and FindSecBugs, the projects being scanned require a fresh build, so a list of build commands for each version of repositories is included in our replication package[7]. Depending on the tool, we export all results to xml or json files.

We followed a manual approach to determining whether a tool accurately detected a vulnerability for two reasons. First, it is not possible to automatically match vulnerable lines with a tool's warnings because the SAP benchmark lacks vulnerable line locations. Second, we were unable to automatically match weakness types, because different tools differ in their classification of vulnerabilities.

To determine whether a tool detected a vulnerability, we match the description from the tool warning with the NVD's description of the vulnerability. We analyse the source code from the vulnerability fixing commit to ensure that the tool warning refers to the same instance of the vulnerability. We use the associated weakness type and description for assistance, but do not rely on the CWE labels, as in many cases a SAST tool labels vulnerabilities differently from the NVD's CWE label. For example, the NVD labels CVE-2014-0107[8] as a CWE-264 vulnerability, but SAST tools that detect this vulnerability label it as a CWE-611.

3.4 Identifying missing patterns from a tool's rule set

To identify detection problems with SAST tools, we examine the vulnerability fixing commit of false negative results. We compare existing tool's rules with the code patterns likely to be vulnerable in order to understand why a vulnerability was missed. Specifically, we performed the following steps for each false negative result:

(1) Determine whether a tool missed a vulnerability due to an insufficient rule, or a missing rule. We use the reported CWE class to filter the existing rules to identify a rule that should have detected the vulnerability but was insufficient.
(2) We then compare the differences between the vulnerability fixing commit and the existing rule's list of patterns to understand why a vulnerability was missed.

```
Iterable<Object> result = new
     org.yaml.snakeyaml.Yaml().loadAll(yaml);
```

Listing 2: Snippet from CVE-2016-8744 fixing commit

As an example, CVE-2016-8744 is a vulnerability reported to the CVE which uses the SnakeYAML library but doesn't use its SafeConstructor by default. Snyk Code detected this vulnerability. FindSecBugs and CodeQL do not have a rule regarding using the SnakeYAML library, whereas Semgrep's existing rule was insufficient in detecting this instance. Listing 1 presents Semgrep's rule for detecting unsafe use of the SnakeYAML library. Listing 2 shows a code snippet from the vulnerability fixing commit of CVE-2016-8744. We compare Semgrep's pattern with the code snippet and identify that Semgrep missed this vulnerability due to its patterns being limited to the initialisation of the SnakeYAML class using the load() method only, and no support for the loadAll() method.

3.5 Configuring Semgrep

We configure Semgrep by incorporating the missing patterns highlighted in RQ3. We use Semgrep's rule editor on its web application[9]. Listing 1 presents an example rule from Semgrep's standard configuration. Semgrep rules require an identifier, affected languages, and patterns, but can also include additional properties such as an accompanying message, or references to external information.

For each identified vulnerable code pattern, we edited an existing rule, or created a new one to include the associated weakness type, a message, references to the CVE/NVD from which the pattern was derived, references to the vulnerability fixing commit, and the pattern(s).

We created 18 new rules and edited 11 existing ones. The newly configured version of Semgrep we refer to as Semgrep* for the rest of the paper. We followed the methodology described in 3.3 with Semgrep* and recorded the results.

4 RESULTS

In this section, we report the results of the SAST tool evaluation, the reasons for the performance, highlight rule improvements, and the outcome of applying these new rules to Semgrep.

RQ1: What is the detection rate of SAST tools?

Table 4 presents the detection rate for each individual tool and the total combined detection rate for the tools. Table 4 shows that

[7] https://github.com/lusastevaluation/sastevaluation
[8] https://nvd.nist.gov/vuln/detail/CVE-2014-0107
[9] https://semgrep.dev/orgs/-/editor

CWE-ID	CWE Name	CodeQL	Snyk	Semgrep	FSB	Combined	#
CWE-611	Improper Restriction of XML External Entity Reference 'XXE'	25.9%	22.2%	48.1%	63.0%	88.9%	27
CWE-502	Deserialization of Untrusted Data	22.2%	11.1%	22.2%	16.7%	44.4%	18
CWE-22	Improper Limitation of a Pathname to a Restricted Directory 'Path Traversal'	40.0%	40.0%	13.3%	46.7%	73.3%	15
CWE-20	Improper Input Validation	0.0%	0.0%	0.0%	0.0%	0.0%	15
CWE-79	Improper Neutralization of Input During Web Page Generation 'Cross-site scripting'	14.3%	7.1%	7.1%	14.3%	14.3%	14
CWE-94	Improper Control of Generation of Code 'Code Injection'	20.0%	0.0%	30.0%	70.0%	70.0%	10

Table 3: The detection rate of each tool broken down into different weakness types. See Table 6 for full list of all weakness types.

FindSecBugs offered the best results of any single tool with a detection rate of 26.5%, whereas Snyk Code detected the fewest vulnerabilities with an 11.2% detection rate. Combining all four tools increases the detection rate to 38.8%.

Tool Name	Detection Rate	No. of Instances
CodeQL	15.3%	26/170
Snyk Code	11.2%	19/170
Semgrep	15.9%	27/170
FindSecBugs	26.5%	45/170
Total	38.8%	66/170

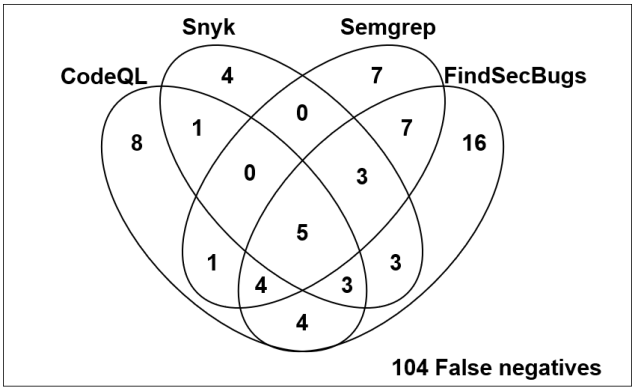
Table 4: The detection rate for the SAST tools and their combined total

Finding 1: SAST tools vary in detection rate from 11.2% to 26.5%, but combining tools improves detection rate to 38.8%.

Table 3 shows the detection rates decomposed by their CWE type. The tools show a range of performance for various vulnerabilities. The detection of CWE-611 vulnerabilities, for example, ranges from 22.2% to 63%, with a combined total of 88.9%. The detection of CWE-611, CWE-22 and CWE-94 vulnerabilities show the best performance. However, CWE-502, CWE-20 and CWE-79 are among the least detected.

Finding 2: SAST tools show good performance at detecting some weakness types (CWE-611, CWE-22, and CWE-94). However, developers using SAST tools should be aware that deserialization of untrusted data (CWE-502), input validation (CWE-20), and cross-site scripting (CWE-79) are among the least detected vulnerabilities. Developers should focus their manual vulnerability detection efforts on these least detected vulnerabilities.

Figure 1: Venn diagram of the detected vulnerabilities.



RQ2: What tool and/or combination of tools detect the most vulnerabilities?

Table 4 shows that the combined detection rate is 38.8% (66/170), which is less than the sum of all tools, suggesting that tools overlap.

Figure 1 illustrates the number of vulnerabilities uniquely detected by each tool and the overlap between the groupings. Figure 1 shows that FindSecBugs detects 16 vulnerabilities that other tools could not detect, the most vulnerabilities uniquely detected by a single tool. The combined total number of vulnerabilities detected by a single tool is 35 (8+4+7+16); 53% (35/66) of detected vulnerabilities were detected by a single tool. In addition, Table 6 highlights those weakness types exclusively detected by a single tool, such as CodeQL, which was the only tool to detect any CWE-1333: Inefficient Regular Expression Complexity vulnerabilities. While tools do show some overlap, a high number of vulnerabilities were detected by a single tool, suggesting that tools are focusing on different things, or differ in their rules. No tool completely subsumes another tool, and, to detect the most vulnerabilities, implementing a range of tools is needed. The five vulnerabilities in the middle of the Venn diagram are detected by all four tools, and they are all different weakness types: CWE-502, CWE-79, CWE-295, CWE-22, CWE-611. This means each tool has rules to detect instances of these vulnerabilities.

Finding 3: The majority of vulnerabilities were uniquely detected by a single tool, SAST tools exclusively detected some weakness types, and no tool fully subsumes another. Therefore, to achieve the best possible performance using the standard configuration, a suite of SAST tools is needed.

RQ3: What is the reason a SAST tool could not detect a vulnerability?

In this section, we analyse the vulnerabilities not detected by SAST tools, discuss the reasons why a tool could not detect the vulnerability, and suggest improvements to the rule set of SAST tools. We first present findings for some weakness types on a high level, then present detailed examples for the top five most frequent weaknesses in our dataset.

There are two main reasons for a SAST tool not being able to detect a vulnerability: missing rules or patterns from the tool's rule set; or a static approach is not suited to detect a vulnerability. Highlighted in the Appendix, Table 6 presents the full list of weakness types and their detection rate for each tool. SAST tools were unable to detect any instance of CWE-835: Infinite loops and CWE-362: Race condition type vulnerabilities, which are due to insufficient control flow management. Dynamic Application Security Testing (DAST) tools may be better suited for insufficient control flow vulnerabilities, as they test for vulnerabilities while software is in operation.

Finding 4: All SAST tools were unable to detect infinite loops and race condition vulnerabilities. Developers should not rely on static analysis as the only method of vulnerability detection.

We go into detail regarding the missing patterns from each tool's rule set and highlight some examples for the five most frequent weakness types in our dataset.

CWE-611

CWE-611: Improper Restriction of XML External Entity Reference ('XXE') occurs when processing specially crafted XML documents that resolve to documents outside the intended sphere of control. The exploitation of XXE vulnerabilities can lead to a denial of service, exposure of information or code execution. Table 3 shows detection rates range from 22.2% to 63.0%, with a combined total of 88.9%. Vulnerable software typically uses third party libraries that handle XML files, such as DocumentBuilder or SAXParser, but are vulnerable to attacks when using the default configuration of these libraries. The most common vulnerable code patterns are an instance of an XML parser without the necessary security restraints and a common fix is to disable features from untrusted sources. The most common vulnerable code patterns are an instance of an XML parser without the necessary security restraints. Listing 3 demonstrates a common fix, which is to disable features from untrusted sources.

```
private static final DocumentBuilderFactory DBF =
     DocumentBuilderFactory.newInstance();

static {
    try {
        DBF.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,
             true);
+       DBF.setFeature(".../disallow-doctype-decl", true);

        DBF.setValidating(false);
        DBF.setIgnoringComments(false);
```

Listing 3: Common fix for a CWE-611

We find the reason for SAST tools' inability to detect some instances of vulnerabilities is due to an incomplete taxonomy of rules regarding third party classes and supporting methods to securely process XML files. For example, FindSecBugs detects the most XXE vulnerabilities, but lacks rules regarding the javax.xml.validation SchemaFactory class and methods of creating javax.xml.parsers SAXParser. We identify improvements to all tools by examining the three instances not detected by any tool involving the org.dom4j.io SAXReader class and creation of the XMLInputFactory class; some examples are highlighted in Listing 4.

```
    // Likely to be vulnerable code patterns for
         new/updated rules
    (...)new SAXReader();
    (...)SAXReader().newInstance();
    (...)XMLInputFactory.newInstance();
```

Listing 4: Missing vulnerable code patterns for CWE-611

Finding 5: With a combined total of 88.9%, CWE-611 is the most detected weakness type by most tools. However, even the best performing tools miss instances of these vulnerabilities involving the javax.xml.validation.SchemaFactory and javax.xml.parsers.SAXParser classes.

CWE-502

CWE-502: Deserialization of Untrusted Data. Deserializing an object without verifying it first can result in a denial of service or code execution. Many of these vulnerable instances were fixed by limiting what types can be deserialized, also known as *white/blacklisting* certain objects. To detect CWE-502 vulnerabilities, SAST tools rely on common Java classes that deserialize objects such as the ObjectInputStream class. Semgrep detects four instances (22.2%) detecting the use of the ObjectInputStream object. However, the combined result is 44.4% with each tool detecting at least one unique instance, revealing vulnerable code patterns unique to each tool. Of the 18 CWE-502 vulnerabilities, 10 could not be detected by any tool and 7 belonged to the Jackson Databind project; while these are catalogued as different vulnerabilities, they are very similar and the fixes involve white/blacklisting new classes. No tool could detect the source of the vulnerability in the Jackson Databind project, however, Semgrep and FindSecBugs have rules to detect

the use of the Databind library itself. The remaining 3 instances not detected highlight a potential vulnerable code pattern: using the com.jmatio.io.MatFileReader class and not disabling object deserialization as seen in Listing 5.

```
// Vulnerable code
com.jmatio.io.MatFileReader mfr = new
     com.jmatio.io.MatFileReader(...);

// Fix
static {
    MatFileReader.setAllowObjectDeserialization(false);
}
```

Listing 5: Vulnerable code pattern for detecting CWE-502 vulnerabilities using the MatFileReader class

CWE-22

CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') attacks aim to access data outside the root folder using the "../" sequence. SAST tools show good performance detecting path traversal vulnerabilities, 73.3% in total with each tool detecting 40% with the exception of Semgrep. Semgrep could only detect 13.3%. The poor performance of Semgrep was due to an attempt to reduce false positives. Listing 6 shows two examples of vulnerable lines, both contain a user input string "image" to create a new file which could contain a "../" sequence. Semgrep detects the first as a potential path traversal vulnerability, whereas the second is not considered vulnerable since the org.apache.commons.io.FilenameUtils.getName(...) method is used. However, we found that using the org.apache.commons.io.FilenameUtils.getName(...) method was insufficient in removing path traversal vulnerabilities and additional checks were required.

```
// Semgrep Match
File file = new File("static/images/", image);

// Semgrep does not match
File file = new File("static/images/",
     FilenameUtils.getName(image));
```

Listing 6: Semgrep's patterns for detecting path traversal vulnerabilities

> **Finding 6:** SAST tools detect most (73.3%) path traversal vulnerabilities. However, in an attempt to reduce false positives, Semgrep's standard configuration misses some instances.

CWE-20

CWE-20: Improper Input Validation type vulnerabilities refer to absent or incorrect validation of user input. The CWE list is hierarchical and CWE-20 is an abstract weakness type; tools may label these vulnerabilities as something more concrete. For example, path traversal is a more specific vulnerability that can follow an improper input validation. We originally had 29 instances of improper input validation vulnerabilities. However, 14 were detected and more accurately labelled by the SAST tools. From the remaining 15 instances that could not be detected, a more suitable label could not be found, nor any new vulnerable code patterns. Fixes to improper input validation vulnerabilities typically involve additional checks to String objects. For SAST tools to detect improper input validation vulnerabilities, context regarding the String objects is required (otherwise, a large number of false positives would be generated).

CWE-79

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting' (XSS)) are a type of injection and occur when software does not neutralize user input. SAST tools show poor performance and could only detect 14.3%. This is surprising, since CWE-79 attacks are severe and often the most common vulnerability on CWE's top 25 most dangerous vulnerabilities lists [1]. SAST tools have several rules regarding CWE-79 vulnerabilities, such as detecting user input in an output stream. However, these seem to be insufficient when detecting XSS vulnerabilities in production code. SATE VI performed an evaluation of seven tools on 30 manually injected CWE-79 vulnerabilities in a Java repository and found tools performed significantly better, with one tool detecting 100% [15]. The discrepancy between test data could suggest a wider variety of vulnerable code patterns in production code than found in synthetic data.

```
// Vulnerable code
... new org.apache.wicket.markup.html.form.TextArea(...)

// Fix
... new org.apache.wicket.markup.html.form.HiddenField(...)
```

Listing 7: Vulnerable code pattern for detecting CWE-79 vulnerabilities and fix

The fixes regarding XSS vulnerabilities were quite different, but two examples made changes to the org apache wicket markup html form TextArea class, Listing 7 shows the potential vulnerable pattern and suggested fix.

While some vulnerable code will be difficult for SAST tools to detect without producing a large number of false positives, many SAST tools miss some "easy to detect" vulnerabilities. We use the identified missing rules to improve Semgrep's standard configuration and present the results in the following section.

RQ4: What is the detection rate for a single tool after configuration?

In this section, we present the results of evaluating the newly configured Semgrep*. Table 5 lists the detection rates of the various tools and their combined total. By editing the configuration of Semgrep, we were able to increase its detection rate from 15.3% (26/170) to 44.7% (76/170), an increase of 181%. Semgrep* detects all the vulnerabilities detected by each tool, and an additional 10 vulnerabilities not detected by any other tool.

> **Finding 7:** By editing the configuration of a SAST tool, the detection rate was increased by 181%, offering a higher detection rate than combining tools.

Semgrep* offers better detection rates than combining multiple SAST tools using their standard configuration. Configuring a single tool offers better detection rates, and produces fewer warnings than using a suite of tools, as it does not produce duplicate warnings between tools.

Tool Name	Detection Rate	No. of Instances
CodeQL	15.3%	26/170
Snyk Code	11.2%	19/170
Semgrep	15.9%	27/170
FindSecBugs	26.5%	45/170
Combined	38.8%	66/170
Semgrep*	44.7%	76/170

Table 5: The detection rate for SAST tools, their combined total, and the newly configured Semgrep*

4.1 Discussion

Recommended SAST usage

FindSecBugs offers the best results from any single tool, detecting 45 vulnerabilities. However, we found that tools were complementary, with 53% of detected instances being unique to a single tool. This suggests that even open source SAST tools use different sets of rules and developers should be aware that their tool of choice cannot detect all vulnerabilities. To use SAST tools effectively and get the best results possible, developers must do one of the following: firstly, understand the underlying vulnerable code patterns of SAST tools and choose an appropriate tool for their code base; secondly, engage with the tool's configuration, adding more custom rules that better suit their code style/guides; or thirdly, use a combination of different tools.

Johnson et al. [20] found that a high number of false positives and developer overload are barriers of use for SAST tools. Following the above suggestions may be too much for developers, which risks leaving code unprotected. Our results show configuring a tool offers better detection rates, and will produce fewer warnings overall due to multiple tools overlapping. Therefore, we recommend that developers using SAST tools configure their chosen tool. Beller et al. [14] found less than 5% of rules from configurable static analysis tools were custom rules. It is unclear whether the understanding and effort to configure a tool's rule set is a deterrent or developers are unaware of the benefits of implementing custom rules.

Effective false positives

SAST tools produce a large number of warnings. However, care should be taken when ignoring warnings. In Finding 6, we report that Semgrep's standard configuration missed vulnerabilities due to ignoring the warning if some safety checks were present, i.e. effective false positives. However, the vulnerability fixing commit suggests these safety checks were insufficient, and additional checks were required to remove the vulnerability. Developers ignoring certain warnings from SAST tools should ensure it is the right decision, as real vulnerabilities may be missed.

Vulnerability detection techniques

Table 3 presents the detection rates of each tool per weakness type. CWE-611 is the most detected weakness for most tools, whereas tools were unable to detect any CWE-835 infinite loop vulnerabilities. Our findings suggest that SAST tools may be more suited for certain weakness types; while improving SAST tools is crucial, other vulnerability detection techniques should be incorporated to detect more vulnerabilities. This is corroborated by Elder et al. [18], who found that SAST tools, DAST tools, and manual analysis techniques were complementary.

Vulnerability labelling

Labelling and classifying vulnerabilities requires expert knowledge of the vulnerability and classification scheme. The CWE is a hierarchical list of weakness types with many overlapping views. SAST tools do not agree on labelling vulnerabilities. Some tools label a vulnerability as a CWE-22: Path Traversal and others label the same vulnerability as a CWE-23: Relative Path Traversal. While this is a small discrepancy, it indicates how difficult this can be for developers who submit vulnerabilities to the CVE. Around half the vulnerabilities in our test data were labelled as a different weakness type by a SAST tool than reported in the NVD. There also seems to be some confusion as to whether to label vulnerabilities based on the cause or consequence. A CWE-611: XXE vulnerability is the cause, but can lead to remote code execution, mistakenly labelled as a CWE-94: Injection. The NVD is the biggest open source of production vulnerable data. While it does require expertise and effort to accurately label a reported vulnerability, it is important for future work, especially as the NVD grows and more automated approaches can be used to compare vulnerable code patterns.

Synthetic Datasets

Until more examples of vulnerabilities are submitted, more effort can be made to introduce a wider variety of code patterns in synthetic datasets. As highlighted in 3, our results show that tools can only detect 14.3% of XSS vulnerabilities, significantly less than previous studies evaluating SAST tools on synthetic data. The SATE VI report [15] manually injected 30 XSS vulnerabilities into code and found tools detected up to 100%. A large difference in detection rates between tools when evaluated on production code and synthetic datasets suggests that synthetic datasets do not fully represent the variety of code patterns found in production code. In addition, a wider variety of weakness types should be present in synthetic datasets. The Juliet Test Suite for Java [3], a popular synthetic dataset, lacks many of the weakness types found in production code. Frequently submitted weakness types, such as CWE-611, CWE-502, CWE-20, CWE-94, are not present in the Juliet Test Suite. This omission prevented us from comparing our results on production code with a synthetic dataset.

4.2 Threats to Validity

Internal validity refers to the validity of results, typically concerning causality. We identified tools from OWASP's list of SAST tools, which may not be a complete list. The selected tools are inconsistent in their CWE labelling; two tools may label the same vulnerability as different CWEs. In addition, the source of our data, the NVD, has been noted to be inconsistent or inaccurate with its CWE labels and GitHub references. Therefore, detection rates could be due to inaccuracies in the data rather than performance of a tool. We mitigate concerns with CWE labelling by manually reviewing the GitHub commit message, NVD description of the vulnerability, CWE description of the weakness, and the message provided by the tool warning to validate the tool could detect the vulnerability.

External Validity refers to the validity of applying conclusions outside the scope of the test data. Our data set is limited in size and biased towards the small sample of contributing projects and repositories. We note a possible reason for low detection rates is that coverage is limited and may contain vulnerability types unsuitable for SAST tools to detect, such as infinite loops and race conditions. Therefore, the detection rates and vulnerable code patterns identified may not be generalisable to a wider software sample. However, we mitigate this threat by not comparing flat detection rates with previous studies, but by breaking the detection rates into individual weakness types.

Construct validity refers to how well a test measures the concept it was designed to evaluate. Each SAST tool produced a large number of warnings, and the manual verification process was performed by a single author. Therefore, valid warnings may have been missed and detection rates may not be accurate. However, we report the detection results of tools not to present the best tool in all scenarios, but to understand why a vulnerability cannot be detected to improve a tool's taxonomy of rules.

5 RELATED WORK

It is important to understand the limitations of SAST tools, and, as such, there are a number of studies evaluating their performance. Previous studies have evaluated static analysis tools on synthetic datasets [11, 12, 15, 17, 19, 21, 22].

For example, Mahmood & Mahmoud [22] evaluated four static analysis tools for both C/C++ and Java on examples from the Software Assurance Reference Dataset (SARD). The study reported which tools could detect 10 different vulnerability types, but did not report how many instances were tested or the detection rate of each weakness type. Synthetic datasets typically contain excellent ground truth and enough examples to ensure results are statistically significant; however, synthetic vulnerable code examples lack realism and do not represent production code, producing inflated detection rates [19, 21]. In our study, we evaluate tools on production code to derive a more accurate depiction of detection rates.

There has been some work evaluating tools on production code. Amankwah et al., [8] evaluated different detection methods including fuzzing, web application scanners, static analysis techniques, Binary Runtime Integer Based Vulnerability Checkers (BRICK) and C Range Error Detectors (CRED); they found the different methods to be complementary. Amankwah et al.'s test data is derived from the NVD, but is limited in size and does not disclose the types of vulnerabilities.

In a study for the DLR German Aerospace Center, Gentsch [19] evaluates tools on both the Juliet Suite for C/C++ and a trial on production software with known vulnerabilities. Gentsch's study evaluated tools for C/C++ and the results may not be applicable to Java. In addition, there is no breakdown in terms of weakness types evaluated, meaning it is difficult to compare detection rates between synthetic datasets and production code.

Li et al. [21] evaluated 7 SAST tools for Java on a manually curated benchmark of production code derived from the NVD. Our study shares a lot of similarities with that of Li et al. Both studies evaluate Java SAST tools on production code, and report on the detection rate, overlap of tools and dissect the reasons for poor performance. Despite our shared source of data (NVD), our datasets only share a small amount of overlap (39/170 vulnerabilities). Li et al.'s study found 76.9% of false negatives were due to missing or inadequate rules. Our study builds on Li et al. by identifying the detection problems in the four tools we evaluate and, most significantly, we edit the configuration of a tool to overcome the missing and inadequate rules.

6 CONCLUSIONS AND FURTHER WORK

In this paper, we ran several SAST tools on production code with known vulnerabilities and analysed the results to discover detection rates; we also investigated why certain vulnerable instances could not be detected. Our results showed that FindSecBugs detected the most vulnerabilities of a single tool, but even the best performing tools miss some "easy to detect" instances due to an incomplete taxonomy of rules. In addition, a large number of vulnerabilities are uniquely detected by a single tool, meaning that developers must understand their SAST tool or combine them to get best results. However, this will add even more difficulty to using SAST tools. We edit or add new rules to Semgrep, a configurable SAST tool, and found that configuring a tool can offer better detection rates than using a combination of tools with their standard configuration. However, future work is needed to understand how developers can be best equipped and supported to configure their SAST tools effectively.

Future work analysing vulnerable code patterns could include using ML and NLP techniques, as currently the amount of correctly labelled data makes this challenging. It is important that developers submitting vulnerabilities to the CVE label the vulnerability appropriately. Our work improving the detection rate of SAST tools can help developers by increasing the number of detected vulnerabilities, decrease the time taken for debugging, and assist in the effort to label vulnerabilities. Our contributions may increase the number, and overall quality of submitted vulnerabilities to allow analysis using ML and NLP techniques.

REFERENCES

[1] 2022 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Accessed: 2023-07-19.

[2] Cve program mission. https://www.cve.org/. Accessed: 2023-08-04.

[3] Nist software assurance reference dataset. https://samate.nist.gov/SARD/. Accessed: 2023-07-19.

[4] Owasp nodegoat. https://github.com/OWASP/NodeGoat. Accessed: 2023-07-19.

[5] Source code analysis tools. https://owasp.org/www-community/Source_Code_Analysis_Tools. Accessed: 2023-07-18.

[6] Static analysis tool exposition (sate). https://www.nist.gov/itl/ssd/software-quality-group/samate/static-analysis-tool-exposition-sate. Accessed: 2023-07-18.

[7] Alhazmi, O., Malaiya, Y., and Ray, I. Security vulnerabilities in software systems: A quantitative perspective. In *IFIP Annual Conference on Data and Applications Security and Privacy* (2005), Springer, pp. 281–294.

[8] Amankwah, R., Kudjo, P. K., and Antwi, S. Y. Evaluation of software vulnerability detection methods and tools: a review. *International Journal of Computer Applications 169*, 8 (2017), 22–27.

[9] Ami, A. S., Moran, K., Poshyvanyk, D., and Nadkarni, A. " false negative–that one is going to kill you": Understanding industry perspectives of static analysis based security testing. *arXiv preprint arXiv:2307.16325* (2023).

[10] Anderson, R. Security in open versus closed systems—the dance of boltzmann, coase and moore.

[11] Anupam, A., Gonchigar, P., Sharma, S., SB, P., and MR, A. Analysis of open source node. js vulnerability scanners.

[12] Arusoaie, A., Ciobâca, S., Craciun, V., Gavrilut, D., and Lucanu, D. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (2017), IEEE, pp. 161–168.

[13] Baca, D., Petersen, K., Carlsson, B., and Lundberg, L. Static code analysis to detect software security vulnerabilities-does experience matter? In *2009 International Conference on Availability, Reliability and Security* (2009), IEEE, pp. 804–810.

[14] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), vol. 1, IEEE, pp. 470–481.

[15] Black, P. E., Cupif, D., Haben, G., Loembe, A.-K., Okun, V., and Prono, Y. Sate vi report.

[16] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering 48*, 9 (2022), 3280–3296.

[17] Delaitre, A., Stivalet, B., Black, P., Okun, V., Cohen, T., and Ribeiro, A. Sate v report: Ten years of static analysis tool expositions, 2018-10-23 2018.

[18] Elder, S., Zahan, N., Shu, R., Metro, M., Kozarev, V., Menzies, T., and Williams, L. Do i really need all this work to find vulnerabilities? an empirical case study comparing vulnerability detection techniques on a java application. *Empirical Software Engineering 27*, 6 (2022), 154.

[19] Gentsch, C. Evaluation of open source static analysis security testing (sast) tools for c.

[20] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 672–681.

[21] Li, K., Chen, S., Fan, L., Feng, R., Liu, H., Liu, C., Liu, Y., and Chen, Y. Comparison and evaluation on static application security testing (sast) tools for java.

[22] Mahmood, R., and Mahmoud, Q. H. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code. *arXiv preprint arXiv:1805.09040* (2018).

[23] McGraw, G. From the ground up: The dimacs software security workshop. *IEEE Security & Privacy 1*, 2 (2003), 59–66.

[24] Morrison, P., Pandita, R., and Xiao, X. Are vulnerabilities discovered and resolved like other defects? In *Empir Software Eng 23* (2018), p. 1383–1421.

[25] Munaiah, N., Camilo, F., Wigham, W., Meneely, A., and Nagappan, M. Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Software Engineering 22* (2017), 1305–1347.

[26] Ponta, S. E., Plate, H., Sabetta, A., Bezzi, M., and Dangremont, C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), IEEE, pp. 383–387.

[27] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Gall, H. C., and Zaidman, A. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering 25* (2020), 1419–1457.

7 APPENDIX

CWE-ID	CodeQL	Snyk	Semgrep	FindSecBugs	Combined	Instances
CWE-611	25.9%	22.2%	48.1%	63.0%	88.9%	27
CWE-502	22.2%	11.1%	22.2%	16.7%	44.4%	18
CWE-22	40.0%	40.0%	13.3%	46.7%	73.3%	15
CWE-20	0.0%	0.0%	0.0%	0.0%	0.0%	15
CWE-79	14.3%	7.1%	7.1%	14.3%	14.3%	14
CWE-94	20.0%	0.0%	30.0%	70.0%	70.0%	10
CWE-863	0.0%	0.0%	0.0%	0.0%	0.0%	9
CWE-835	0.0%	0.0%	0.0%	0.0%	0.0%	8
CWE-287	0.0%	0.0%	0.0%	20.0%	20.0%	5
CWE-200	0.0%	0.0%	0.0%	0.0%	0.0%	5
CWE-295	25.0%	25.0%	25.0%	25.0%	25.0%	4
CWE-352	0.0%	0.0%	25.0%	25.0%	50.0%	4
CWE-1333	75.0%	0.0%	0.0%	0.0%	75.0%	4
CWE-284	0.0%	0.0%	0.0%	0.0%	0.0%	3
CWE-297	0.0%	0.0%	0.0%	0.0%	0.0%	2
CWE-601	50.0%	0.0%	50.0%	50.0%	50.0%	2
CWE-362	0.0%	0.0%	0.0%	0.0%	0.0%	2
CWE-77	0.0%	0.0%	0.0%	0.0%	0.0%	2
CWE-74	0.0%	50.0%	0.0%	0.0%	50.0%	2
CWE-269	0.0%	0.0%	0.0%	0.0%	0.0%	2
CWE-119	0.0%	0.0%	0.0%	0.0%	0.0%	2
CWE-521	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-208	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-319	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-755	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-532	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-113	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-78	0.0%	0.0%	0.0%	100.0%	100.0%	1
CWE-190	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-918	0.0%	100.0%	0.0%	100.0%	100.0%	1
CWE-330	0.0%	0.0%	0.0%	100.0%	100.0%	1
CWE-770	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-400	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-643	0.0%	0.0%	0.0%	100.0%	100.0%	1
CWE-91	0.0%	0.0%	0.0%	0.0%	0.0%	1
CWE-1188	0.0%	100.0%	0.0%	100.0%	100.0%	1

Table 6: Detection rates and the full breakdown of CWE-IDs