Survey of Enhanced Coverage Metrics: Do Software Static Analyzers Look at Everything?

Andrew D. Barlow
Jason Garcia Solorzano
Jeffrey Reginald
Chinyere Sloley
Charles Nicholas
Carolyn Seaman

College of Engineering and Information Technology
University of Maryland Baltimore County

Abstract

Current software static analysis tools are designed to analyze a given codebase (e.g. a software product, app, or library) to provide detailed indications of potential software vulnerabilities, either malicious or unintentional. These analyzers help developers and users to identify and mitigate weaknesses in the code. What the analyzers generally do not provide is information about how thoroughly they examined the codebase being analyzed. This information would be particularly useful for users trying to determine the suitability of a codebase for acquisition or inclusion into a software product. The SoECM project explores the issue of test code coverage of static analysis tools. We evaluated 10 widely available static analyzers to determine their level of coverage, and to design a way to enhance these tools to better report their coverage metrics. We found evidence that all the analyzers achieve 100% code coverage, although we cannot be completely confident in that result for the commercial analyzers we evaluated. We also built a simple prototype that illustrates how an analyzer could be instrumented to report coverage information.

1. Introduction

Current software static analysis tools are designed to analyze a given codebase (e.g. a software product, app, or library) to provide detailed indications of potential software vulnerabilities, either malicious or unintentional. These analyzers help developers and users to identify and mitigate weaknesses in the code. They are often used to evaluate code that is being considered for acquisition or reuse. One of the missing outcomes of most static analyzers is an indication of what was and was not actually evaluated for the analysis report. If a tool finds many (real)

weaknesses, the user has some confidence that it is thorough. But if few weaknesses are reported, does that mean there aren't many weaknesses or did the tool not analyze some parts of the code it was applied to? To instill confidence in the software being analyzed, there needs to be a mechanism to report the actual coverage of static analysis tools. Coverage includes what code segments or basic blocks or modules were examined, as well as what types of vulnerabilities (buffer overflow, memory leakage, SQL injection, hardcoded passwords, etc.) were considered. Such information would allow the reviewer or integrator or user to perform a more thorough assessment of risks related to the incorporation of the software into their environment as part of the supply chain. Parts of the codebase that were not examined by the static analyzer would present higher risk.

The aim of this study is to 1) determine, for a given sampling of static analysis tools, the actual coverage achieved, and 2) explore ways to capture coverage information and present it to the user as part of the output of the static analyzer, and produce a prototype showing how this could be done.

2. Related work

Some previous studies have evaluated and compared static analysis tools in terms of their underlying machine learning based algorithms, the accuracy of the results and cost. To examine accuracy, many studies examine static analysis results in terms of precision, recall and performance. This is possible when a static analyzer is applied to a codebase that contains known vulnerabilities that the analyzer is expected to catch. Precision is calculated by dividing the number of True Positives (TP), by the sum of True Positives and False Positives (FP), i.e. $P = \frac{TP}{TP + FP}$. Where TP is the number of vulnerabilities that were correctly identified, and FP is the number of vulnerabilities reported by the analyzer that were not actual vulnerabilities. Recall is calculated by dividing TP by the sum of TP and FN (the number of false negatives, i.e. the number of known vulnerabilities that the the analyzer did not report), i.e. $R = \frac{TP}{TP + FN}$". (Kulenovic & Donko, 2014) Finally, the cost of different static analysis approaches is evaluated as the “measure of computing resources needed to generate the results”. (Anderson, 2018) These studies revealed that “the most effective tool is one that strikes the right balance among false positives, false negatives, and performance”. (Kulenovic & Donko, 2014) Also, static analyzers tend to have high cost when applied to code containing recursive functions, therefore the test code is key to evaluating and comparing static analyzers in terms of accuracy and cost.

The most commonly evaluated static analyzer in published studies is FindBugs. However, other static analyzers evaluated include: ITS4, SPLINT, UNO, Checkstyle, ESC/Java, FindBugs, PMD (Li & Cui, 2010) Yasca, FindBugs, Microsoft Code Analysis Tool.Net (Cat.Net) (AlBreiki & Mahmoud, 2014), Gimpel PC Lint, PVS - Studio, Red Lizard Goanna Studio, and CppCheck (O.V. et al., 2014), Parfait, (Kulenovic & Donko, 2014), VUDENC (Wartschinski et al., 2022), SonarCloud Vulnerable Code Prospector for C (Raducu et al., 2020). Most past studies exercised the analysis tools by running them on specific test code. While this is a controlled way of evaluating a static analyzer, Lipp et al. (2022) examined the differences in static analysis

results when analyzing test code versus real world code and found a significant disparity. No past studies, to our knowledge, have examined static analysis tools to determine the source code coverage they provide.

3. Methods

In order to evaluate static analysis tools for their level of coverage, we first had to choose the analyzers to include in our work. We then had to choose test code to apply those analyzers to. Finally, we followed a defined process to exercise both commercial and open source static analyzers to determine their level of code coverage.

a. Choosing static analyzers to evaluate

We used the following criteria to choose the static analyzers for our study:
(i) Open-source or commercial and if commercial, a free or community version is available.
(ii) Detection of SQL Injection and Buffer Overflow vulnerabilities.
(iii) Available documentation to help determine its coverage and functionality.

We found static analyzers by examining papers from the literature describing previous studies of static analyzers, and investigating the appropriateness for our work of the analyzers used in those studies. We also used, as a starting point, a list of static analyzers provided by our collaborator at NIST. All source code and tools used were hosted in the Discovery, Research, and Experimental Analysis of Malware (DREAM) Lab at UMBC. We ended up with a set of 10 static analysis tools used for this study, which are shown in Table I.

Name	OSS or commercial?	Link
SonarQube	Both	https://github.com/SonarSource/sonarqube
Coverity Scan	Commercial	https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
SonarLint	Open Source	https://plugins.jetbrains.com/plugin/7973-sonarlint#JetBrains
CodeScene	Commercial	https://codescene.com/community-edition
CodeQL	Commercial	https://codeql.github.com/
CodeHawk-C	Open Source	https://github.com/static-analysis-engineering/CodeHawk-C
Cppcheck	Open Source	https://github.com/danmar/cppcheck
Bandit	Open Source	https://github.com/PyCQA/bandit
Programming Mistake Detector (PMD)	Open Source	https://github.com/pmd/pmd
DevSkim	Open Source	https://github.com/Microsoft/DevSkim

Table I: List of static analysis tools evaluated.

Most of these tools are open source projects available to the public. CodeScene, Coverity Scan, and SonarQube offer a commercial version as well as a limited open source version through cloud-based services.

b. Choosing test code

Our goal during this research was to analyze the coverage of a collection of static code analysis tools meant to find security vulnerabilities. Our criteria for choosing our test code was that it contains known buffer overflow, SQL injection, and OS command injection vulnerabilities, which are common vulnerabilities that have been prevalent in attacks in recent years. According to the Common Weakness Enumeration (CWE), a community developed list of common software and hardware weaknesses that is maintained by The MITRE Corporation, SQL and OS command injection attacks are listed at number 3 and 6 respectively in their 2022 CWE Top 25 Most Dangerous Software Weakness list.[1] The datasets and repositories chosen are all free and open-source software.

The aforementioned security vulnerabilities can be seen in various programming languages such as C, C++, Python, and Java. The first place that we searched for a dataset of vulnerable code was the National Institute of Standards and Technology (NIST). NIST's Software Assurance Reference Dataset (SARD) is a growing collection of almost two hundred thousand test programs with documented weaknesses. From the SARD's dataset, we chose the test suite Juliet C/C++ version 1.3,[2] which contains about 64,099 test cases including buffer overflow, OS command injection attacks, and many more. The SonarCloud Vulnerable Code Prospector for C (SVCP4C) dataset is a collection containing buffer overflow vulnerabilities in more than 10,000 source code files written in C.[3] The seeve repository is another dataset with 30 programs written in C that cover 20 CWEs.[4] Vulpy is a web application developed in Python, Flask, and SQLite that was intentionally designed to have vulnerabilities.[5] The Open Web Application Security Project (OWASP) Benchmark Project is a Java test suite that contains 2,740 test cases and

[1] CWE (2022). 2022 CWE Top 25 Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html
[2] NSA Center for Assured Software (2017). Juliet C/C++ (Version 1.3) [Source Code]. https://samate.nist.gov/SARD/test-suites/112
[3] Raducu, R., Esteban, G., Rodríguez Lera, F. J., & Fernández, C. (2020). Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation. Applied Sciences, 10 (4), 1270. DOI: https://doi.org/10.3390/app10041270
[4] Conikeec (2019). Seeve [Source Code]. https://github.com/conikeec/seeve
[5] Portantier, Fabian Martinez (2020). Vulpy - Web Application Security Lab [Source Code]. https://github.com/fportantier/vulpy

covers 11 CWEs.[6] Other vulnerable basic OS command injection[7] and SQL injection [8] files were used for initial testing before moving on to datasets with larger quantities of files.

c. Evaluation process

Once the static analyzers to be evaluated had been chosen, and appropriate test code chosen, we then proceeded to evaluate each static analyzer, one by one, to determine their level of test code coverage. The process we followed for this evaluation differed slightly depending on whether the analyzer being evaluated was an open source tool or a commercial tool.

For open source analyzers, the process required that we first download and install the tool under investigation, and explore the functions implemented by the tool by reading the accompanying documentation and exercising the user interface. The we reviewed the open-source code to identify key components and understand the functionality related to code coverage. Finally, we set up a testing infrastructure to run the analyzer on the test code and record the vulnerabilities found as well as measure the coverage of the test code. We compared the vulnerabilities found to the vulnerabilities known to be in the test code and tried to reason about any differences. We measured coverage by examining the report of modules that the analyzer provided (in some cases) or by instrumenting the code of the analyzer to report this information.

For commercial analyzers, we first attempted to find a free version of the analyzer, or in some cases a web-based interface that provided some of the functionality of the tool. We then installed the free version (or created an account with the web-based interface) and explored the analyzer's functions by exercising the interface. We then had to prepare test code to be submitted to the tool, which was in some cases non-trivial. We then ran the analyzer on the sample code and recorded the vulnerabilities found. We then compared the list of vulnerabilities found to those known to exist in the test code. This allowed us to reason about coverage, both in terms of vulnerability types found and code coverage.

4. Results

a. Source code coverage of tools

As expected, we were able to gather much more information about code coverage with the open source static analyzers than the commercial tools. We were in fact able to confirm that all the open source analyzers we examined do cover all of the test code they are given to analyze. Some go even further, for example CPPCheck attempts to cover all possible executions, based on all combinations of compiler directives.

---

[6] OWASP (2016). OWASP Benchmark (Version 1.2) [Source Code]. https://owasp.org/www-project-benchmark/
[7] Zhong, Weilin. Command Injection [Source Code]. https://owasp.org/www-commun
ity/attacks/Command_Injection
[8] Pankaj (2022). SQL Injection in Java and How to Easily Prevent it.
https://www.digitalocean.com/community/tutorials/sql-injection-in-java

However, not all the OSS analyzers found all the known vulnerabilities in the test code they analyzed, for a variety of reasons. For example, CPPCheck attempts to find the least number of false positives as possible, so it does not report things it is not highly confident in. Thus, it did not report many of the buffer overflow vulnerabilities in the test code. Although we focused on buffer overflow and SQL injection vulnerabilities in our analysis, some of the analyzers we evaluated were not designed to detect these types of problems. CPPCheck, DevSkim, and CodeHawk-C were not designed to detect SQL injection vulnerabilities. CodeQL did not look for SQL injection vulnerabilities in Python code. PMD is not able to detect any security vulnerabilities because it has no security rulesets.
In a few cases, there were indications that the analyzers we were examining were finding the buffer overflow and SQL or command injection vulnerabilities we were looking for, but we could not be sure. For example, we used Bandit to analyze a Python based web application that was intentionally designed to have vulnerabilities. Bandit covered all of the code that it was tested on, but we could not come to a conclusion if all possible SQL and command injection attacks were detected because although Bandit flagged many SQL injections, we could not determine how many vulnerabilities were present in the Python project. DevSkim found poor uses of functions that would cause a buffer overflow except for those of sprintf().DevSkim would sometimes report it had found something without stating the name of the function, and we suspect those cases could possibly be pointing to the sprintf()function.

Our results for the commercial tools we evaluated were similar, although we had less information available to understand the inner workings of these tools. All the commercial tools we evaluated appeared to cover all the test code they were given to analyze. However, none of them found all the vulnerabilities that were known to be in that test code. There are a variety of reasons for this, including the ones discussed above for OSS analyzers. In most cases, the reasons for failing to find known vulnerabilities appeared to more likely be related to the intended capabilities of the analyzer, rather than a failure to achieve full code coverage. For example, CodeScene focuses on development time, maintenance cost, and helping with code review and is not designed to detect any buffer overflow, SQL injection, and OS command injection vulnerabilities. Coverity Scan reported buffer overflow and command injection vulnerabilities, but not nearly as many as were known to be in the test code. This does not appear to be a code coverage problem, as Coverity Scan reported covering all the code, but we were not able to determine the reason that it missed so many vulnerabilities. It could be that Coverity Scan uses a higher threshold for reporting vulnerabilities, to avoid too many false positives, as described above for CPPCheck. SonarQube's free community edition (which we used for this study) could detect the known SQL and OS command injection vulnerabilities, but not buffer overflow vulnerabilities because it does not analyze C and C++ projects. Finally, SonarLint was able to scan the entire project we ran it on, but it could not detect the known vulnerabilities it contained.
In summary, we have high confidence that all the OSS static analyzers we evaluated achieved 100% coverage of the test code they analyzed, and good confidence that the commercial analyzers did so as well. However, this does not mean that all the analyzers find all the vulnerabilities in the code that they analyze. We found a number of reasons why the vulnerabilities we knew to be present in the test code were not all found, including what types of

vulnerabilities the analyzer was designed to detect, the thresholds used by the analyzer to determine if an issues is vulnerable “enough” to report, and the rulesets used by the analyzer.

b. Prototype solution to report coverage information

The second research task in this project is to create a prototype demonstrating how an analyzer might report code coverage information to the user. Our prototype involves the modification of CPPCheck (version 2.8), an open-source static analyzer previously touched upon. CPPCheck was chosen for modification due to its largely friendly structure of file parsing and rule checking. In order to successfully analyze source code contained in a file for the purpose of static analysis, the analyzer in question must construct some kind of data structure to store the file content. In this case, CPPCheck creates an abstract syntax tree. After this tree is constructed, each part of it is compared against some basic, default rules provided by the developers. Some of these rules include “checkmemoryleak”, “checknullpointer”, and “checkbufferoverrun”. There are 27 in total, which are all based on one or more CWEs. Some of these rules are implemented in XML files as patterns, and some are implemented as basic string comparisons. We implement our prototype by modifying the code that carries out this comparison process to make information about coverage visible to the user.

These rules, henceforth known as “checks”, are encapsulated within a class known as Check. Because of this, we are able to easily output which check the current chunk of test code is being compared against. As an aside, it is important to understand that CPPCheck will attempt to analyze all possible combinations of preprocessor directives given a source code file to analyze. When explicitly printing which check is run against which source code file, one would assume there to be roughly $n * 27$ lines of output related to just checks, where n is the number of files to check. However, because of CPPCheck’s attempt to analyze all possible combinations, there will always be at least $n * 27$ lines. For example, with our instrumentation of CPPCheck, running the tool on a dataset of 11,422 vulnerable C files, we would expect approximately 308,394 lines of output related to checks. However, a full run has yielded 990,927.

We are able to verify that every combination of preprocessor directives is checked by examining the number of combinations yielded per file. When instrumented to do so, we find that CPPCheck is able to successfully parse and tokenize around 146,500 configurations of preprocessor directives, all containing different combinations of potentially vulnerable source code. As a result, we know that the various checks have been compared against at least 146,500 variations of the dataset. Such information, when included in the output, has the potential to vastly increase one’s confidence in a static analyzer.

Further analysis of the output of CPPCheck shows that it does tend to hang during execution in some threads. As a result, only 11,414 out of the 11,422 total files (~99.93%) were able to be fully analyzed. Figure 1 shows an example screenshot of the prototype output.

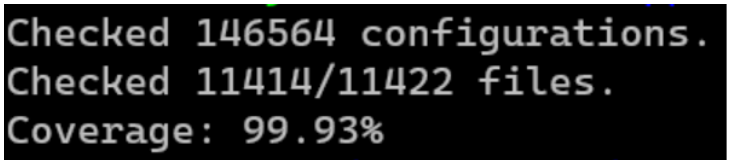

Figure 1: Sample output of code coverage prototype.

5. Conclusions

The SoECM project had as a goal to explore a representative sample of software static analysis tools to learn about their level of code coverage (i.e. the amount of the code being analyzed that was actually examined by the static analyzer) and to find a way to report the level of coverage that the analyzers achieve. To that end, we examined a total of 10 static analyzers that are widely available. Most were available open source, but a few were commercial tools. We found that all the tools appeared to achieve 100% code coverage of the test code submitted to them, although we could not be completely confident in that result for some of the commercial tools. We also found variation among the tools in terms of what vulnerabilities were identified and reported. Many analyzers did not report all the vulnerabilities known to be present in the test code, for a variety of reasons including the design of the analyzer, the rules followed to identify vulnerabilities, and the thresholds used to determine the significance of vulnerabilities found. However, test code coverage provided by the analyzer was not found to be a source of the differences in vulnerabilities found. We also developed a simple prototype to report coverage by instrumenting one of the open source analyzers with code that reported the components being analyzed, and then calculated the code coverage achieved.

There are a variety of useful and interesting avenues for future work. One is to enhance our simple prototype to be more robust and to be applicable to different static analyzers. Another is to further explore the coverage of a wider variety of static analyzers. This exploration could examine code coverage more deeply, but it could also examine and catalog differences in how different analyzers identify vulnerabilities. We found differences in our results that point to variation among static analyzers in the logic used to identify vulnerabilities, the thresholds used to determine if a potential issue constituted a vulnerability, and the types of vulnerabilities that each analyzer was designed to detect. Understanding and documenting these differences would be a useful contribution to the field.

References

Anderson, D. P. (2018, June). The use and limitations of static-analysis tools to improve software ... Retrieved September 28, 2022, from https://www.researchgate.net/publication/215835966_The_Use_and_Limitations_of_Static-Analysis_Tools_to_Improve_Software_Quality

H. H. AlBreiki and Q. H. Mahmoud, "Evaluation of static analysis tools for software security," 2014 10th International Conference on Innovations in Information Technology (IIT), 2014, pp. 93-98, doi: 10.1109/INNOVATIONS.2014.6987569.

Lipp, S., Banescu, S., & Pretschner, A. (2022). An empirical study on the effectiveness of static C code analyzers for vulnerability detection. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. https://doi.org/10.1145/3533767.3534380

M. Kulenovic and D. Donko, "A survey of static code analysis methods for security vulnerabilities detection," 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014, pp. 1381-1386, doi: 10.1109/MIPRO.2014.6859783.

O.V., P., & D.O., I. (2014). Making static code analysis more efficient. Journal of Cyber Security and Mobility, 3(1), 77–88. https://doi.org/10.13052/jcsm2245-1439.315

Peng Li and Baojiang Cui, "A comparative study on software vulnerability static analysis techniques and tools," 2010 IEEE International Conference on Information Theory and Information Security, 2010, pp. 521-524, doi: 10.1109/ICITIS.2010.5689543.

Raducu, R., Esteban, G., Rodríguez Lera, F. J., & Fernández, C. (2020). Collecting vulnerable source code from open-source repositories for dataset generation. Applied Sciences, 10(4), 1270. https://doi.org/10.3390/app10041270

Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., & Grunske, L. (2022). VUDENC: Vulnerability detection with deep learning on a natural codebase for python. Information and Software Technology, 144, 106809. https://doi.org/10.1016/j.infsof.2021.106809