Comparing ML-Based Predictions and Static Analyzer Tools for Vulnerability Detection*

Norbert Vándor, Balázs Mosolygó, and Péter Hegedűs

Department of Software Engineering, University of Szeged, Hungary
{vandor|mbalazs|hpeter}@inf.u-szeged.hu

Abstract. Finding and eliminating security issues early in the development process is critical as software systems are shaping many aspects of our daily lives. There are numerous approaches for automatically detecting security vulnerabilities in the source code from which static analysis and machine learning based methods are the most popular. However, we lack comprehensive benchmarking of vulnerability detection methods across these two popular categories. In one of our earlier works, we proposed an ML-based line-level vulnerability prediction method with the goal of finding vulnerabilities in JavaScript systems. In this paper, we report results on a systematic comparison of this ML-based vulnerability detection technique with three widely used static checker tools (NodeJSScan[1], ESLint[2], and CodeQL[3]) using the OSSF CVE Benchmark[4]. We found that our method was more than capable of finding vulnerable lines, managing to find 60% of all vulnerabilities present in the examined dataset, which corresponds to the best recall of all tools. Nonetheless, our method had higher false-positive rate and running time than that of the static checkers.

Keywords: Vulnerability detection, ML models, Static analyzers, OSSF benchmark

1 Introduction

Software security is becoming more and more crucial as software systems are shaping many aspects of our daily lives. A seemingly minor programming error

* This research was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme and the framework of the Artificial Intelligence National Laboratory Program (MILAB). Furthermore, Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences and the ÚNKP-21-5-SZTE-570 New National Excellence Program of the Ministry for Innovation and Technology.

[1] https://github.com/ajinabraham/nodejsscan
[2] https://eslint.org
[3] https://codeql.github.com
[4] https://github.com/ossf-cve-benchmark/ossf-cve-benchmark

might turn out to be a serious vulnerability that causes major losses in money or reputation, threatens human lives, or allows attackers to stop vital services or block infrastructure. Therefore, finding and eliminating security issues early in the development process is very important.

Since security testing is very hard and requires lots of expertise and has high costs, automated solutions are highly desirable. There are numerous approaches for detecting security vulnerabilities in the source code from which static analysis [20] and machine learning based methods [11] are the most popular. Even though tools are compared to each other within their categories (static analysis tools with other static analysis tools or ML models with other ML models), we lack comprehensive benchmarking of vulnerability detection methods across categories.

Empirical comparison of ML based vulnerability detection and static analysis tools might bring in useful insights that could help determining if one of the technologies are more beneficial than others, are there categories of security issues that can be detected more effectively with certain technique, or is combining these techniques bring anything performance-wise.

In one of our earlier works [18], we proposed an ML-based line-level vulnerability prediction method with the goal of finding vulnerabilities in JavaScript systems, while being both granular and explainable. Since our method provided favorable results, it was the next natural step to see how it fares against other static analyzers. Therefore, in this paper we report results on a systematic comparison of this ML-based vulnerability detection technique with three widely used static checker tools (NodeJSScan, ESLint, and CodeQL) for identifying vulnerabilities.

As a benchmark, we selected the OpenSSF (OSSF) CVE Benchmark[5] that consists of code and metadata for over 200 real life CVEs[6] It also contains tooling to analyze the vulnerable codebases using a variety of static analysis security testing (SAST) tools and generate reports to evaluate those tools. We extended the benchmark with the integration of our own ML-based prediction tool and evaluated the results on the data contained in the benchmark.

We found that our method was more than capable of finding vulnerable lines, managing to find 60% of all vulnerabilities present in the examined dataset. We also showed that it is capable of finding vulnerabilities that other tools would likely miss, as it has a higher likelihood of finding issues belonging to CWEs[7], which the others find significantly less of. Nonetheless, our method had higher false-positive rate and running time than that of the static checkers.

The rest of the paper is organized as follows. Section 2 describes our ML-based methods, the static analyzer tools and the benchmark we used for comparison. We provided detailed results on the 200 CVEs contained in the benchmark in Section 3. We present related literature in Section 4 and enumerate the possible threats to our work in Section 5. Finally, we conclude the paper in Section 6.

[5] `https://github.com/ossf-cve-benchmark/ossf-cve-benchmark`
[6] Common Vulnerabilities and Exposures, `https://www.cve.org`
[7] Common Weakness Enumeration, `https://cwe.mitre.org`

2 Approach

Our main goal in this paper is to compare how well a JavaScript line-level ML-based vulnerability prediction method works when compared to classical static analyzer tools for vulnerability detection. Therefore, in this section, we describe the essence of an ML-based vulnerability detection method [18] we developed for identifying vulnerable JavaScript code lines and the benchmark we used for comparing it to other static analysis vulnerability checkers.

2.1 The VulnJS4Line Method

The method we have created aims to detect vulnerable lines in code bases by comparing each one to a preexisting database of known vulnerabilities. The process can be broken up into 2 major parts:

The first part is the creation of the knowledge base, and word2vec[8] model, that will be used to create vectors from the lines in question. This step needs to be executed only once, since the knowledge base can easily be extended after its creation, and the model does not need to be retrained.

The second part is the actual prediction process, during which each line of the system under investigation is tested against each line of the vulnerability database, in order to find the "most similar" vulnerable line. The probability of a line being vulnerable is calculated based on its distance from its closest pair in the database, and some static rules, such as the lines' complexity. If this probability exceeds a certain threshold, we mark it as vulnerable.

During the creation of the results presented in this paper, we used the parameters we found to be most successful during our initial testing.

2.2 OSSF CVE Benchmark

To evaluate our tool, we used the OSSF CVE Benchmark. This project consists of code and metadata for over 200 real life CVEs (belonging to 42 CWEs) from over 200 GitHub projects, while also providing the required tooling to test the performance of static analyzers. Each project contains one CVE and the patch for it; and each CVE affects one file. In total, there are 223 affected files, which translates to 222752 lines of code. For proper testing, it is invaluable to use examples from real life, rather than synthetic test code.

How it works. The benchmark uses drivers to run the different tools on all CVEs, both pre- and post-patch, then determines two things: 1) was the tool able to detect the vulnerability, or did it produce a false negative? 2) when ran against the patched codebase, did it recognize the patch, or did it produce a false positive? After the run, all drivers generate reports, which then the benchmark parses and presents as an HTML or text file. In the report made by the benchmark, one can see certain statistics, such as the number of correctly identified

[8] `https://radimrehurek.com/gensim/models/word2vec.html`

vulnerabilities and the number of correctly recognized patches. Besides these, there is also the file containing the vulnerabilities, with icons denoting which tool flagged which line and for what reason.

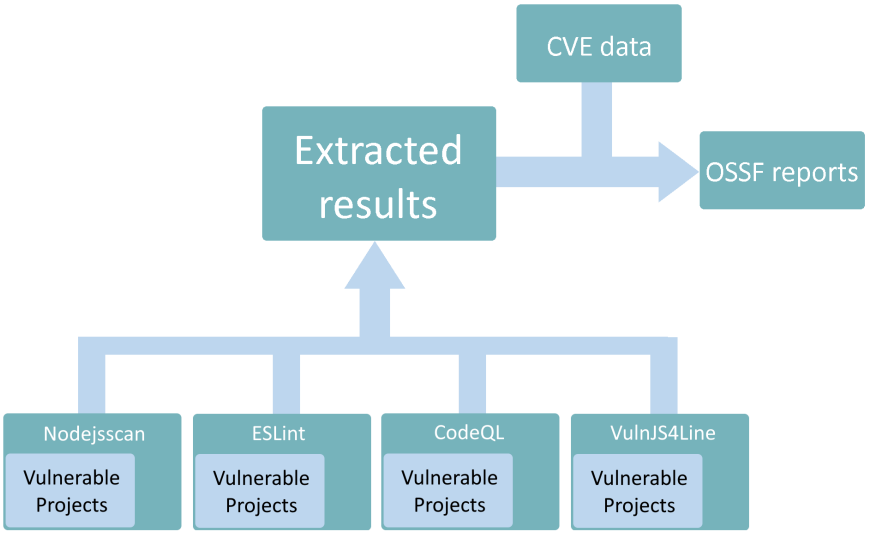


Fig. 1: A vizualization of the benchmarking process

VulnJS4Line driver. For our own tool, we had to write a driver file in TypeScript, whose main function is to run the tool, collect the output, and produce a report compatible with the benchmark. First, the driver runs our tool on the project folder the benchmark has provided for a specific CVE. To be able to do this, we had to upgrade our tool to accept folders as input, and search for and check all JavaScript files inside them. For easier handling of outputs, we also added the option to get the results as a JSON file. To get the best results, we set the parameters to those that we outlined during our previous work. In other words, we used the SpecialValues model with 92% using the prefer_complex rule.

After this, the driver collects the results and creates an output JSON file. The files read by the benchmark must contain pairs of data: the first element is the rule the CVE violates, and the second element is a location consisting of the path to the file and the line the vulnerability is in. The rules are, in other words, the reason why a tool flags a certain line. For us, it's the lexed version of the line the CVE is the most similar to. And finally, the output file is passed to the benchmark, which stores it.

2.3 Other Tools in the Benchmark

We used three different static analyzers to which we compared our method. All of them, including the necessary drivers, were provided by the benchmark.

NodeJSScan is an open source static code scanner used to find security flaws specifically in Node.js applications. It is a user interface built upon njsscan[9], a command-line interface that uses regex-based pattern matching to find bugs and enforce code standards.

ESLint [10] is a static code analysis tool for identifying problematic patterns found in JavaScript code. It can be used for both code quality and coding style issues. It supports the current standards for ECMAScript, and is also able to analyze TypeScript code. ESLint is the most commonly used JavaScript linter.

CodeQL [11] is a semantic code analysis engine, which lets users query code as it was data. First, CodeQL has to create a relational database from each source file in the codebase by monitoring compiler activity and extracting relevant information in the case of compiled languages, or simply running the extractor directly on the soruce code to resolve dependencies in the case of interpreted languages. The database contains a full hierarchical representation of the code, including the abstract syntax tree, the data flow graph, and the control flow graph. Then it uses queries to find specific issues in the code. It achieves that by performing variant analysis, which is the process of identifying "seed vulnerabilities" - in other words, an already known security vulnerability - and using them to find similar problems in the code.

2.4 Statistics for the Comparison

As presented in Section 3, the three main statistics we were interested in were prediction quality, the time required to analyze a project and the most detected vulnerability types. To get the necessary data for these statistics, we had to extract some extra information that is not normally used in the reports the benchmark generates. While the reports contain a list of lines the tools flagged as vulnerable, to create the statistics regarding the average number of flagged lines, we also had to get the total number of lines in a file. Fortunately, this is included in the source of the benchmarking tool, along with important information regarding the CVEs.

After getting the average statistics, the next step was to create the time statistics. Unfortunately, the benchmark does not provide a conventional way to measure the time required for each tool. To remedy this, we analyzed the console logs produced by the benchmark. Before running a tool, the benchmark prints a crucial piece of information for us: the timestamp for when it will shut a

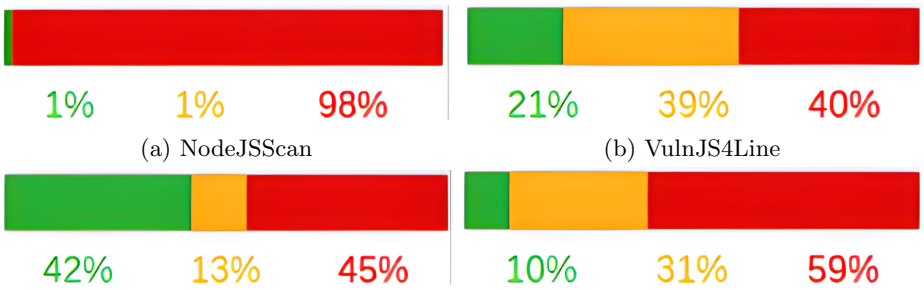
[9] `https://github.com/ajinabraham/njsscan`
[10] `https://eslint.org`
[11] `https://codeql.github.com`

running tool down. In other words, since we know that a tool will timeout after exactly 30 minutes, we could measure how long it ran based on the previously mentioned timestamps.

For the third statistic, as mentioned, we did have all information regarding CVEs, including their CWEs, which we were interested in. However, to determine whether a tool properly found a certain CWE, we also needed to check if it flagged a relevant line - this information is provided in the reports. So to create this statistic, we simply counted how many relevant flags were produced for each CVE, and from that we could check which CWE it belongs to.

3 Results

In this section, we will discuss the performance of our approach compared to three well-established and widely used static analysis tools, when it comes to detecting vulnerabilities. We have selected 3 metrics to showcase here in order to properly demonstrate the potential of our approach. These metrics are what we believe to be the most important when it comes to usability.



(a) NodeJSScan (b) VulnJS4Line

(c) CodeQL (d) ESLint

Fig. 2: Visualization of the tools' performances

3.1 Prediction Quality

When it comes to usability, the most important property of a vulnerability detection tool is its ability to produce accurate results.

As mentioned in 2 the OSSF CVE Benchmark checks whether a method flags a vulnerable lines before and after it has been patched or not. In an optimal scenario, a method would only mark actually vulnerable lines as vulnerable, however that is unfortunately not the case in most situations. In many cases the tools detect potentially faulty lines based on characteristics that do not change after the actual issue has been fixed, and as such continue to flag the lines as vulnerable, even after it has been dealt with.

The results shown in Figure 2 demonstrate the above point. Percentage of perfect cases, where the line is only marked before a fix has been issued is represented in green, while orange represents cases where a line has been marked both before and after the related issue has been dealt with. Naturally, red represents the lines that were never marked as vulnerable.

We can see, that our method finds the most lines overall, while perfectly handling the second most lines, outperforming both NodeJSScan and ESLint by a significant margin when it comes to both metrics. CodeQL finds less vulnerable lines overall, however stops flagging most of them after they have been fixed.

We consider our results to be the second best, when it comes to this measure of predictive quality, since our method correctly handles more than twice as many vulnerabilities as ESLint does. However, the amount of "orange" or partially handled vulnerabilities is a cause for concern, since it is a sign of a higher false positive rate. Technically, in this metric, our method is outdone by NodeJSScan, since that tool fully handles half of the lines it finds, meaning that only half of the vulnerabilities will be marked after they have been fixed. We still would not consider this to be the second best result in this area, because the low probability of NodeJSScan actually finding a vulnerability would still lead to a user that solely depends on the tool missing a large majority of issues.

A similar metric, unfortunately omitted by the benchmarking tool, is the amount of lines flagged by each tool overall. The tool only measures the false positive rate of the methods in question in relation to finding vulnerabilities, but not their overarching performance, when it comes to marking lines. This means, that a method, that flags every line, would achieve a 100% "orange" rating.

As we can see in Figure 3 on average, our method flags the most lines, followed by ESLint. The average examined file is approximately 1000 lines long, getting a bit longer post patch, meaning that while our method performs the worst, it still only flags about 10% of all lines. While this is a major issue when it comes to usability, since a large false positive rate can render a tool completely useless in a commercial setting where development time is a key asset, with manageable improvements, our tool could compete with a widespread solution such as ESLint.

3.2 Time Requirement

The effectiveness of a vulnerability detection tool can be greatly improved, if it is runnable in real time, during the process of writing code, or even in a just in time fashion, after a commit is created. The sooner a developer is notified about a potential issue, the easier it is to double check the results, and discard false positives. Even if a method is capable of producing high quality results, a massive time requirement can still hinder its large scale adoption. This metric is unfortunately omitted from the OSSF CVE Benchmark.

As we can see in Figure 4 our method takes 6 minutes on average to create a set of results, while the overall best performing tool CodeQL takes a little over a minute. This is a major blow to the possibility of practical use, when it comes to our method. Improvements could still be made, since the tested version was

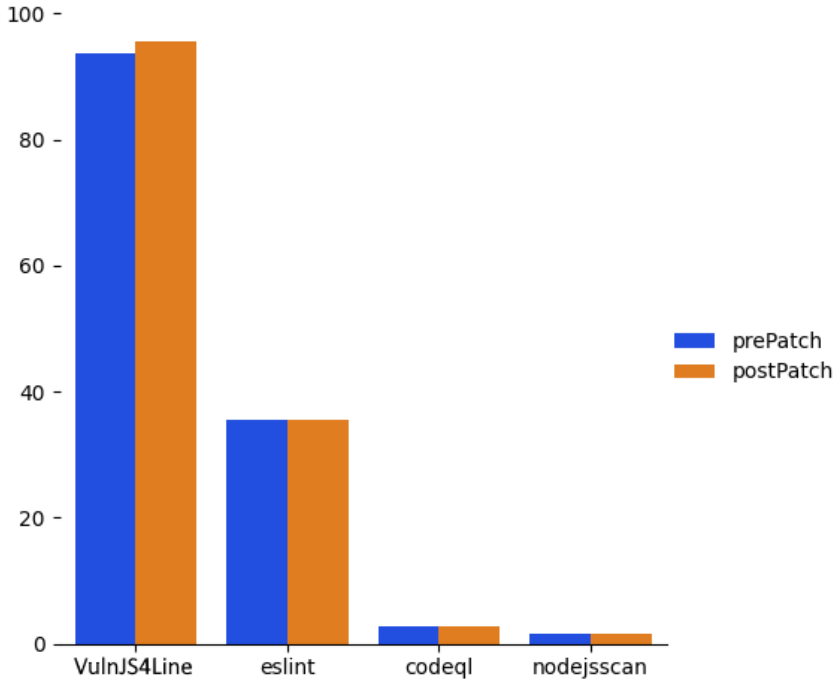


Fig. 3: Average number of lines marked per file

written in Python, a language generally known for its poor performance, when it comes to computation speed. Since the theoretical complexity of our method is low, it is possible, that with major technical improvements, it could become competitive, with its widely used counterparts.

3.3 Most Detected Vulnerability Type

Related literature [2,10,13,14] agrees that static analyzers are performing better when it comes to finding SQL Injection and XSS related vulnerabilities. We investigated, whether our method showed similar preferences, when it came to finding different vulnerability types, or not.

As mentioned in Section 2.4 the Benchmark we used contains the CWE[12] categories of the vulnerabilities in question.

In Table 1 we showcase the CWEs that showed significant performance gaps between the tools, while having over 10 occurrences. Our main goal with presenting these results is not to show differences in the tools' capabilities, rather to point out potential areas, where one may be preferable. For this reason, we

[12] Common Weakness Enumeration

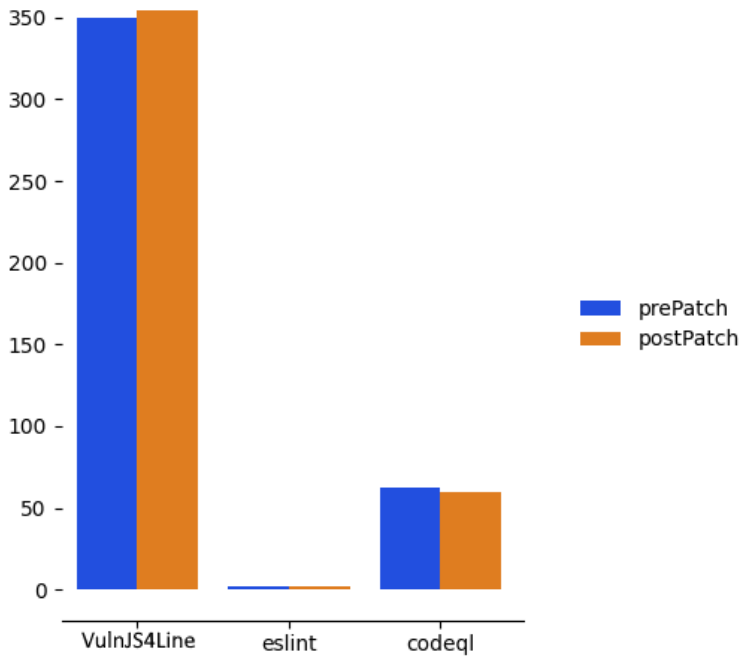


Fig. 4: Average runtime of methods in seconds

omit information about vulnerability types that rarely appear, or those where the tools' performance is similar.

We do not showcase the results produced by NodeJSScan, since it marks too few lines, for its performance to be comparable, when it comes to finding different vulnerability types.

SQL Injection and Cross Site Scripting. Despite our notion of only showcasing CWEs with a relevant presence, we include CWE-94 even though it appears only 4 times in the dataset. CWE-89 is the id given to SQL Injection related vulnerabilities, and as such its inclusion is warranted.[13] We can see, that none of the tools manage to find a significant number of lines containing issues of this category.

Cross-site scripting or XSS related vulnerabilities, with the CWE id of 79, also do not present outstanding numbers, in this context.[14] We can see, that out of the portrayed tools, even the highest performing one, does not reach 50% detection rate.

[13] https://cwe.mitre.org/data/definitions/89.html
[14] https://cwe.mitre.org/data/definitions/79.html

	CWE-78	CWE-79	CWE-88	CWE-89	CWE-94	CWE-116	CWE-400	CWE-730	CWE-915
VulnJS4Line	70%	37%	72%	25%	58%	22%	56%	55%	65%
CodeQL	40%	44%	44%	25%	35%	52%	32%	27%	35%
ESLint	55%	40%	16%	0%	78%	17%	62%	27%	95%

Table 1: The amount of CWEs found by three of the tested tools

In view of the above mentioned results, we conclude, that in this context, the tools do not perform as previous research would indicate. As seen in Section 4.2 these 2 vulnerability types are the ones that are generally found with the highest success rate.

3.4 Key Differences Between the Observed Performance of the Methods

Our method performs outstandingly well, when it comes to CWE-78 (OS Command Injection), CWE-88 (Argument Injection) and CWE-730 (vulnerability related to A9 Denial of Service). It is important to point out, that while in all of these cases, our method manages to find over half of all vulnerabilities present, while the other tools struggle to get above the half way mark, except for ESLint, in the case of CWE-78. In the case of CWE-730, our method manages to find over twice as many vulnerable lines as the other two, while in the other 2 cases, it manages to gather over 70% of the lines related to the issue at hand.

When it is not the highest performing approach, it still does not fall too far behind, providing similar results, to at least one of the tools.

We can also see, that in multiple cases one tool clearly outperforms the rest, especially in the cases of CWE-915 and CWE-116. This is effect has been previously observed in different environments, and leads to th suggestion of using multiple solutions, that have clear, differing strength in tandem, to achieve an overall boosted performance.

Since out method performs distinctly better in areas, the other examined tools falter, it might have place as a specialist tool, when it comes to finding certain kinds of issues.

4 Related Work

Numerous works have been created to either compare vulnerability scanners, check the effectiveness of static analyzers, and to check the validity of benchmarks.

4.1 Effectiveness of the Static Analyzers

V. Benjamin Livshits and Monica S. Lam [15] formulated a general class of security errors in Java programs. Using these they were able to create a precise and

scalable static analysis tool that was capable of finding security vulnerabilities such as XSS, SQLI, HTTP split attacks and more.

Nathaniel Ayewah et al. [6] created and evaluated FindBugs. A static software analysis tool capable of not only finding generic bugs, but security vulnerabilities such as SQL injections.

Katerina Goseva-Popstojanova and Andrei Perhinschi[12] evaluated 3 static analysis tools in the context of C/C++ and Java. They found that a significant portion of vulnerabilities was missed by all three tools, and less then half of the vulnerabilities were found by all three, in both environments.

Elisa Burato et al. [7] were able to create Julia, a static analyzer tool capable of finding 90% of security issues present in the OWASP benchmark, greatly outperforming previous scanners.

4.2 Comparing Vulnerability Scanners

S. El Idrissi et al. [13] found, during their evaluation of multiple vulnerability scanners, that the performance of each scanner differs for different vulnerability types. They found, that the scanners were more effective at finding SQLI and XSS related issues.

Jose Fonseca et al. [10] compared the performance of three commercially used scanning tools, when it came to finding SQLI and XSS related issues. They found that the tested methods produced high false positive rates and relatively low coverage.

Chanchala Joshi and Umesh Kumar Singh [14] also found, that vulnerability scanners are more successful at detecting SQLI and XSS related faults in code.

Mansour Alsaleh et al. [1] examined multiple open source web vulnerability scanners using an approach that allowed evaluation from multiple angles. They found that, while overall there were only minor differences between the performance of each scanner, there was considerable variance when it came to both the type and number of detected vulnerabilities in each tools' results.

Nataša Šuteva et al. [22] tested 6 web vulnerability scanners on the WackoPicko app. The scanners found similar amounts of vulnerabilities, all with a high false positive and false negative rates.

Balume Mburano and Weisheng Si [17] compared not only the performance of different scanners, but 2 different benchmarking tools as well. They found, that using multiple benchmarks to evaluate scanners provides a more accurate picture of their performance. They also found that no scanner can be considered as an all-rounder, since each performs differently in for different vulnerability types.

Andreq Austin and Lauries Williams [5] found that automated penetration testing found more vulnerabilities per hour than static analysis tools, however manual testing was still more effective.

Nuno Antunes and Marco Vieira [2] also investigated the differences between penetration testing and static code analysis. They found, that when it came to detecting SQL Injection related vulnerabilities static analyzers outperformed

their counterparts. They also note, that even tools implementing the same approach to finding issues, might present different results.

Malaka El et al. [9] benchmarked automatic vulnerability detection tools for SCADA devices and scientific instruments. They found that not only did the tools find different vulnerability types with differing efficiency, but their scalability also varied. They propose the idea of using multiple analysis tools in tandem, in order to achieve better results.

4.3 Benchmarks

Creating powerful bechmarking tools even in itself is a challenge.

Valentin Dallmeier and Thomas Zimmermann [8] proposed a technique that allows the automatic collection of successes and failures from a projects history in order to create a large set of benchmarking data.

Reza M. Parizi et al. [16] created a set of guidelines for benchmarking projects.

Nuno Antunes and Marco Vieira [3][4] created a benchmark for web services focusing on SQL Injection related vulnerabilities.

Ivan Pashchenko et al.[19] created a benchmark that aims to reproduce real world scenarios through the use of automatically generated test cases based on prior vulnerability fixes in Open Source software repositories.

Hui-zhong Shi et al. [21] propose a generic framework of Web security evaluation.

5 Threats to Validity

Our evaluation depends on the quality of the data contained in the OSSF benchmark. However, it is manually collected and adopted by other researchers as well, therefore, it poses minor threats to the validity of our conclusions.

We compare a single ML-based method with three different static analysis tools. Therefore, our conclusions might not generalize to other ML-based techniques. Nonetheless, it already gives a first insight into how these different techniques compare to each other for vulnerability detection. However, further studies in this area is needed.

The only evaluation metrics integrated into the OSSF Benchmark are whether a tool marks the vulnerable code lines before the vulnerability is fixed and if they stop reporting it after the fix. However, if a tool marks all lines of a program as being vulnerable, it would get perfect score for the first metric. Therefore, meaningful comparison needs additional evaluation metrics. To mitigate this, we implemented and extended the benchmark measurements with the count of the number of flagged lines as well as the time required for detecting a vulnerability.

6 Conclusion

In this paper we presented the benchmarking results of a tool we created for the purpose of creating line level, explainable vulnerability predictions. For this

purpose we used the OSSF CVE Benchmark, that not only provided an opportunity to test our method in a realistic environment, but to directly compare its capabilities to other, widely used, industry level tools.

We found that our method was more than capable of finding vulnerable lines, managing to find 60% of all vulnerabilities present in the examined dataset. We also showed that it is capable of finding vulnerabilities that other tools would likely miss, as it has a higher likelihood of finding issues belonging to CWEs, which the others find significantly less of.

However, it is limited by two factors. One is its high computation time, taking significantly longer to run, than any of the other examined tools. The other, is its high false positive rate. It flags over twice as many lines as its competitors.

In the future, we plan on reducing the time it takes for the method to run its checks, by heuristically reducing the number of examined lines, and possibly re-implementing it in a language more fit for fast calculations. Before these steps, it is most important to reduce its false positive rate, since we see that as the main limiting factor.

References

1. Alsaleh, M., Alomar, N., Alshreef, M., Alarifi, A., Al-Salman, A.: Performance-based comparative assessment of open source web vulnerability scanners. Security and Communication Networks **2017** (2017)
2. Antunes, N., Vieira, M.: Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In: 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing. pp. 301–306 (2009). https://doi.org/10.1109/PRDC.2009.54
3. Antunes, N., Vieira, M.: Benchmarking vulnerability detection tools for web services. In: 2010 IEEE International Conference on Web Services. pp. 203–210 (2010). https://doi.org/10.1109/ICWS.2010.76
4. Antunes, N., Vieira, M.: Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples. IEEE Transactions on Services Computing **8**(2), 269–283 (2015). https://doi.org/10.1109/TSC.2014.2310221
5. Austin, A., Williams, L.: One technique is not enough: A comparison of vulnerability discovery techniques. In: 2011 International Symposium on Empirical Software Engineering and Measurement. pp. 97–106 (2011). https://doi.org/10.1109/ESEM.2011.18
6. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D., Penix, J.: Using static analysis to find bugs. IEEE Software **25**(5), 22–29 (2008). https://doi.org/10.1109/MS.2008.130
7. Burato, E., Ferrara, P., Spoto, F.: Security analysis of the owasp benchmark with julia. Proceedings of ITASEC **17** (2017)
8. Dallmeier, V., Zimmermann, T.: Extraction of bug localization benchmarks from history. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. p. 433–436. ASE '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1321631.1321702, `https://doi.org/10.1145/1321631.1321702`

9. El, M., McMahon, E., Samtani, S., Patton, M., Chen, H.: Benchmarking vulnerability scanners: An experiment on scada devices and scientific instruments. In: 2017 IEEE International Conference on Intelligence and Security Informatics (ISI). pp. 83–88 (2017). https://doi.org/10.1109/ISI.2017.8004879
10. Fonseca, J., Vieira, M., Madeira, H.: Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In: 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007). pp. 365–372 (2007). https://doi.org/10.1109/PRDC.2007.55
11. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys (CSUR) **50**(4), 1–36 (2017)
12. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Information and Software Technology **68**, 18–33 (2015)
13. Idrissi, S., Berbiche, N., Guerouate, F., Shibi, M.: Performance evaluation of web application security scanners for prevention and protection against vulnerabilities. International Journal of Applied Engineering Research **12**(21), 11068–11076 (2017)
14. Joshi, C., Singh, U.K.: Performance evaluation of web application security scanners for more effective defense. International Journal of Scientific and Research Publications (IJSRP) **6**(6), 660–667 (2016)
15. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX security symposium. vol. 14, pp. 18–18 (2005)
16. M. Parizi, R., Qian, K., Shahriar, H., Wu, F., Tao, L.: Benchmark requirements for assessing software security vulnerability testing tools. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). vol. 01, pp. 825–826 (2018). https://doi.org/10.1109/COMPSAC.2018.00139
17. Mburano, B., Si, W.: Evaluation of web vulnerability scanners based on owasp benchmark. In: 2018 26th International Conference on Systems Engineering (ICSEng). pp. 1–6 (2018). https://doi.org/10.1109/ICSENG.2018.8638176
18. Mosolygó, B., Vándor, N., Antal, G., Hegedűs, P., Ferenc, R.: Towards a prototype based explainable javascript vulnerability prediction model. In: 1st International Conference on Code Quality, ICCQ 2021. pp. 15–25 (2021)
19. Pashchenko, I., Dashevskyi, S., Massacci, F.: Delta-bench: Differential benchmark for static analysis security testing tools. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). pp. 163–168 (2017). https://doi.org/10.1109/ESEM.2017.24
20. Pistoia, M., Chandra, S., Fink, S.J., Yahav, E.: A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM Systems Journal **46**(2), 265–288 (2007). https://doi.org/10.1147/sj.462.0265
21. Shi, H.z., Chen, B., Yu, L.: Analysis of web security comprehensive evaluation tools. In: 2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing. vol. 1, pp. 285–289 (2010). https://doi.org/10.1109/NSWCTC.2010.72
22. Suteva, N., Zlatkovski, D., Mileva, A.: Evaluation and testing of several free/open source web vulnerability scanners (2013)