J-WAVE: A Java Web Application for Vulnerability Education

Michael Alexander Kyer, Virginia Polytechnic Institute and State University
Prof. Stephen H Edwards, Virginia Polytechnic Institute and State University

Stephen H. Edwards is a Professor and the Associate Department Head for Undergraduate Studies in the Department of Computer Science at Virginia Tech, where he has been teaching since 1996. He received his B.S. in electrical engineering from Caltech, and M.S. and Ph.D. in computer and information science from The Ohio State University. His research interests include computer science education, software testing, software engineering, and programming languages. He is the project lead for Web-CAT, the most widely used open-source automated grading system in the world. Web-CAT is known for allowing instructors to grade students based on how well they test their own code. In addition, his research group has produced a number of other open-source tools used in classrooms at many other institutions. Currently, he is researching innovative methods for giving feedback to students as they work on assignments to provide a more welcoming experience for students, recognizing the effort they put in and the accomplishments they make as they work on solutions. The goals of his research are to strengthen growth mindset beliefs while encouraging deliberate practice, self-checking, and skill improvement as students work.

Dr. Bob Edmison, Virginia Polytechnic Institute and State University

Dr. Edmison is a Collegiate Associate Professor in the Department of Computer Science at Virginia Tech. His research focuses on alternative grading practices, accessibility, and the tools to support them.

J-WAVE: A Java Web Application for Vulnerability Education

Abstract

Static application security testing (SAST) tools are commonly used by professionals to identify security vulnerabilities before deployment. While such tools are indispensable in industry, their fragmented ecosystems and complex configuration requirements often preclude their effective use in an educational context. This paper presents J-WAVE (the Java Web Application for Vulnerability Education), a unified web-based framework that encapsulates five industry-standard SAST tools: PMD, FindSecurityBugs, Semgrep, Yasca, and SonarQube. By internalizing tool configuration and providing a scalable REST API capable of processing batch submissions, J-WAVE transforms reactive testing into a proactive pedagogical instrument. J-WAVE offers simplicity to users by handling each tool's setup internally, while offering access to the large, collective rule set contributed by the combined tool suite. Students can scan their own projects easily, while educators can scan many submissions in batch. This paper reports on experiences from applying J-WAVE's tool suite to student submissions in two courses: an advanced data structures course, and a web application development course. Our findings reveal that the integrated tools are highly complementary and that detection efficacy is optimized by tailoring tool prioritization to specific project domains—emphasizing code quality scanners for general applications and vulnerability-focused tools for web environments. This work enables integrating robust, multi-tool security audits into the computer science curriculum.

1 Introduction

In the rapidly expanding field of software development, there is a significant concern for application security. In 2022, a record 25,227 software vulnerabilities were reported as part of the Common Vulnerabilities and Exposures (CVE) database initiative to publicly document security flaws in programming [1]. This figure represents a 25% increase over the vulnerabilities reported in 2021, and 2023 is projected to set another record. In addition to the CVE, the Open Worldwide Application Security Project (OWASP) is a foundation created to advocate for the security of web applications. One initiative developed by the OWASP Foundation is the OWASP Top 10, which aims to provide a current consensus as to the 10 most critical security risks of web applications [2]. These tools combined provide easy access to early vulnerability education.

Unfortunately, it is difficult to apply that education solely by reading the data produced by these projects. The OWASP list is subject to change, and tens of thousands of vulnerabilities are added to the CVE every year. To address this problem, application security testing tools were developed. The term "application security testing" is an umbrella term including static application security

testing (SAST), dynamic application security testing (DAST), interactive application security testing (IAST), and software composition analysis (SCA) [3]. Each type of tool is used at different times within the development process to identify vulnerabilities. A limitation of these tools is that they are a reactive solution for vulnerability detection. In other words, for a vulnerability to be detected, it must be programmed into the application in the first place. Thus, there are potential scenarios in which a vulnerability is introduced into a deployed application without security testing. A better solution would be the combination of reactive application security testing with a proactive solution.

This proactive solution is introduced through the Java Web Application for Vulnerability Education, or J-WAVE. J-WAVE combines the idea of vulnerability education with the practical experience of application security testing. At its core, J-WAVE is the encapsulation of multiple SAST tools used to audit student code written in Java in search of vulnerabilities. The focus on Java exists for several reasons. The first is that Java is a common language in both the classroom and industry, so the experience gained translates well. The second is that exploits found within Java are universal across platforms due to Java's commitment to portability. As a result, Java vulnerabilities can be exploited on any Java-capable machine.

The goal behind the development of J-WAVE was to create a simple solution for vulnerability detection that can be used in the classroom. Each SAST tool has its own unique method for configuration, scanning, and reporting. Setup for each tool can range from importing it into an integrated development environment (IDE), running it from the command line, using the tool from a Docker container, or leveraging a graphical user interface (GUI) provided by the tool's distributor. J-WAVE simplifies this process by running the scans for each tool internally, eliminating the setup required by the user. Additionally, as J-WAVE is a culmination of multiple SAST tools, each one has different detection capabilities. Each tool contains a different set of rules that it uses to detect vulnerabilities. Therefore, certain rules in one tool might not be present in another, and specific tools might be more effective at detecting certain types of vulnerabilities than others.

Both the simplicity and broad detection capabilities of J-WAVE make it ideal for classroom use. Students can easily use J-WAVE to audit their own Java applications and correct them prior to deployment or submission. Experience with J-WAVE is akin to experience with each SAST tool included within. Reports are generated by each tool, and these reports can then be individually parsed to provide key information regarding discovered vulnerabilities. After observing the violations within each report, students can use that information to correct their code. This experience addresses the problem of vulnerability education by providing practical experience with specific vulnerabilities early on during student development. Educators can also use this tool to more effectively teach students about common vulnerabilities. Submitted code can be processed individually or in batches to gain a better understanding of common vulnerabilities. That information can then be used to more effectively teach students how to detect, correct, and prevent instances of these vulnerabilities in the future. In either case, a proactive approach to vulnerability prevention is taken by educating students early on.

Through testing, it was found that J-WAVE could operate on ZIP archives of sizes within the tens of megabytes. These archives can contain thousands of Java files with a maximum tested size of over 9,000 files. This capability is highly beneficial for professors looking to scan section-wide

applications. It was also found that the SAST tools used in J-WAVE are complementary, and using them together can detect a wider range of problems. Certain tools were significantly better at detecting bugs and common design flaws in source code, often referred to as code smells, compared to security vulnerabilities. Other tools were found to be superior at vulnerability detection. To evaluate J-WAVE's utility, we scanned all submissions for five different student assignments, along with one instance of production code. Most vulnerabilities were detected in web applications, along with a large number of bugs and code smells. Student-written data structure projects (non-web applications) almost exclusively reported bugs and code smells. Therefore, depending on the project, different tools should be prioritized.

This paper describes our experiences developing J-WAVE and exploring its applicability. Section 2 addresses prior work related to SAST tools and identifies the list of tools considered for use in J-WAVE along with the reason for including or excluding certain tools. Afterwards, the design of J-WAVE is discussed in Section 3, which addresses how J-WAVE is run, its architecture, and how each scan is performed. After developing J-WAVE, we applied it to an existing production codebase as well as student-written software projects submitted to five different assignments across two courses. Section 4 describes the results of these code scans and the lessons learned regarding the utility of J-WAVE in an educational setting. Each tool's individual performance is analyzed, and conclusions are drawn as to when each tool should be used. The takeaways and implications of J-WAVE's development are summarized in Section 5, with the problems experienced described in Section 6.

2 Related Work

There are multiple articles published and studies conducted that analyze various SAST tools and their applications, as well as some that compare them to one another. One such paper, titled "Developer Companion: A Framework to Produce Secure Web Applications," sought to create a tool that would be directly integrated into an IDE [4]. This tool would detect vulnerabilities in real time. Their study used the FindSecurityBugs SAST tool as the starting point, and they scanned a variety of real-world programs with it. One example of the programs scanned was WebGoat, another project developed by OWASP used to test the validity of emerging SAST tools. They concluded that providing real-time assistance for the developer was the best way to mitigate vulnerabilities and suggested integrating other tools like PMD, Yasca, FindBugs, and LAPSE into their framework. Their ideal solution is a program that provides a suite of tools to scan and detect vulnerabilities in real time.

Sinhabahu et al. [5] aimed to create a tool focused on the visualization of vulnerabilities present in scanned code. They related code to buildings and classes to cities, indicating the presence and severity of vulnerabilities by changing the colors of the visual elements. The SAST tool used as the detection mechanism for these vulnerabilities is SonarQube, and its reports are used to construct the city visualization. While it only used one tool for its scans, the city visualization aspect was shown to be extremely beneficial to developers.

In other work, Sabatini [6] compared a variety of SAST tools in their ability to detect vulnerabilities specifically relating to the Insecure Object Deserialization Vulnerability. The tools used included FindSecurityBugs, SonarLint, SonarQube, ErrorProne, and Semgrep. It found that

FindSecurityBugs and Semgrep were the most effective tools for detecting this specific type of vulnerability, as they were the only tools that detected the flaw within the eight applications scanned. Two of the applications used were WebGoat and a lab created to specifically highlight the deserialization vulnerability. The one exception to the results is that SonarQube could detect some of the vulnerabilities present in the aforementioned lab, but none in any of the other projects.

Many different application security testing tools have been created with the goal of identifying security vulnerabilities, making it difficult to leverage them all. To provide guidance on selecting a tool, the National Institute of Standards and Technology (NIST) created a list of SAST tools to help with vulnerability detection [7]. J-WAVE includes a suite of some of the listed tools, but there were many that were not used for several reasons. Most of the tools listed were ignored as they were unable to scan Java code, or they were cost-prohibitive.

Of the tools available, PMD, FindSecurityBugs, Semgrep, Yasca, and SonarQube were selected for inclusion in the J-WAVE suite. PMD [8] is an open-source project that focuses on identifying code smells, bugs, and vulnerabilities. FindSecurityBugs [9] is an extension of SpotBugs, which itself is an extension of FindBugs. The tool was built to comply with vulnerabilities of both the OWASP Top 10 and the CVE. This tool contains all of the SpotBugs functionality of code smell and bug detection, but with added rules to locate security vulnerabilities. The crucial difference between FindSecurityBugs and PMD is its concentrated focus on vulnerabilities and its scan of bytecode as opposed to source code. Semgrep has a strong focus on vulnerabilities similar to FindSecurityBugs and is also focused on complying with the OWASP Top 10 and other high-level security vulnerabilities [10]. J-WAVE uses the Semgrep Community scan, which contains a curated set of community-developed rules. Enhanced scans can be performed with Semgrep Teams, which includes additional rules developed by Semgrep. However, Semgrep Teams is priced per user in an organization. J-WAVE includes Semgrep Community as it is a free option that does not require account creation, but pricing options can be explored to include these additional rules. Yasca by Michael Scovetta was the fourth SAST tool included within the J-WAVE tool suite. This tool is no longer in active development, but its last update was recent enough to remain relevant [11]. Yasca has an equal focus on vulnerability, code smell, and bug detection. The final SAST tool of the J-WAVE suite is SonarQube. The purpose of this tool is similar to PMD and Yasca in that code smells, bugs, and vulnerabilities are all reported. As a self-managed scanning tool, SonarQube hosts the results from each scan within its own platform [12].

There were also multiple tools within the NIST article that were considered but not integrated. SAST tools Codiga and Deepsource were not used as they required a repository connection to a version control system (VCS). GitLab SAST is the combination of SpotBugs and Semgrep, tools already included in J-WAVE. Jlint and RSM were also considered but were outdated. Options such as Attackflow, AppScan, Reshift, and Snyk code were also considered but not integrated at the time of writing. Each of the previously listed tools appears to be free of charge and is able to scan Java code. Other paid or Java-incompatible options can be found within the NIST article.

3 Designing J-WAVE

J-WAVE is a Java servlet application run on Tomcat. We have created a Docker image encapsulating it for easy deployment. When run, J-WAVE provides a REST API for analyzing files. Requests can be sent via an HTTP POST to the `/main-servlet` endpoint of the user's localhost. J-WAVE is built to handle ZIP archives as input. Once a request is received, the archive is unpacked. J-WAVE then scans each input file with each of its SAST tools (PMD, FindSecurityBugs, Semgrep, Yasca, and SonarQube). The reports of each scan are collected and compressed into a ZIP archive sent as the API's response.

The first two tools integrated, PMD and FindSecurityBugs, relied on downloading their source code and identifying an entry point into the scanning functions. For PMD, an entry-point function, `runPMD`, was invoked that accepted a custom `PMDConfiguration` object loaded with information on how to run the scan. The project, called `PMDRunner`, was then compiled into a JAR archive and placed within the J-WAVE application. When running for the first time, J-WAVE creates a custom class loader that is stored until the application's closure. After ensuring the class loader's creation, the `PMDRunner`'s constructor is invoked with the input directory as a parameter. The integration of FindSecurityBugs was similar to PMD, with slight differences. The entry-point function called was `FindBugs.runMain`, the custom object was called `FindBugs2`, and the compiled project was called `FindSecurityBugsRunner`. In either case, the constructor calls a helper function to set up the custom scanner object and runs the scan using that object.

The third tool integrated was Semgrep. Two solutions for integration were available. The first was a Docker container that runs a Semgrep scan on a mounted directory. There were issues with the Docker container failing to run scans on the given directory, so the other solution was used in J-WAVE. Semgrep was installed via Python's package installer, `pip`, into the J-WAVE container as opposed to running a separate Semgrep Docker container. With the package installed, J-WAVE uses Java's `Process` class to run the Semgrep command similar to how it would be run using a command line interface (CLI).

Yasca was the fourth tool integrated. As the tool is slightly outdated, a custom Docker image was created that has a pre-built environment capable of running Yasca already set up. J-WAVE creates the Yasca child container by preemptively mounting the host's Docker daemon within the J-WAVE container. Another Java `Process` is created to run the container. Within this process, the directory containing the files to scan is mounted, and the report file is copied back over to the J-WAVE container. The created image was pushed to DockerHub for general use.

To run SonarQube, two separate containers are invoked alongside the core J-WAVE container by using Docker Compose. The first container runs the SonarQube community image, which includes a GUI that allows for account and project creation, as well as displaying the scan results for each project. The second container hosts a PostgreSQL database which allows account, project, and scan data to persist across container destruction. A configuration file and local volume directories were created to remove the need for any first-time setup. When invoked, J-WAVE creates a child container to run the scan which links to the SonarQube host to send the results. J-WAVE then requests the scan results from the SonarQube host and sends the response to an output file.

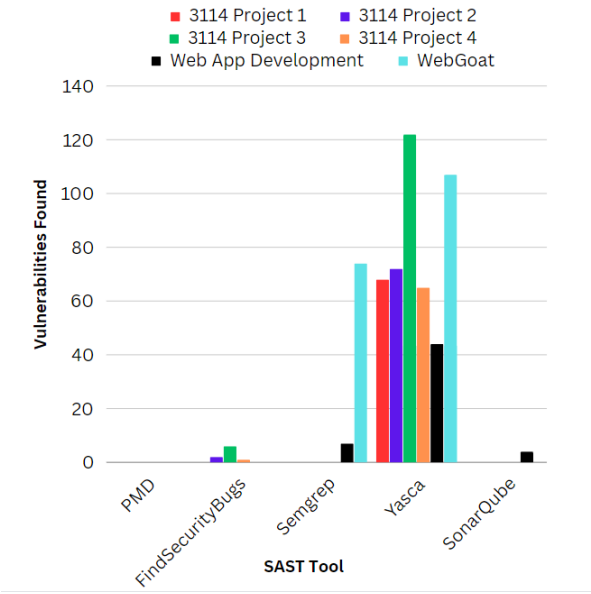

Figure 1: Vulnerabilities found by each SAST tool.

4 Results

By testing J-WAVE, we sought to determine whether the tool is capable of scanning student code and to compare each tool. For professors, it is useful to scan an entire collection of submissions. Therefore, it must be demonstrated that J-WAVE is capable of accepting thousands of files at once and that each tool properly scans every file. It is also important to understand when certain tools should be prioritized over others and whether a tool produces consistent, meaningful output.

In the testing of J-WAVE, six different projects were passed through in their entirety for scanning. Four of these projects came from the undergraduate "Data Structures and Algorithms" course (CS 3114) at Virginia Tech. An archive of the final submissions for each student was processed by J-WAVE. Similarly, an archive of student submissions for the graduate level "Web Application Development" course was also used as input. Finally, the open source project discussed earlier, WebGoat, was used.

Figure 1 depicts the number of vulnerabilities found by each tool for each project. Notably, PMD detected zero vulnerabilities. The main reason for this is that PMD has only two rules to detect security flaws. Many of the SAST tools used detect more than just security flaws. The bulk of problems found are code smells, while a smaller amount are bugs. Vulnerabilities or security flaws are a much smaller minority of the reported issues. Figure 2 shows the proportion of issues found by each tool that were vulnerabilities versus code smells.

To obtain an accurate representation of true vulnerabilities, certain filters must be applied when analyzing the data. The charts compare the number of vulnerabilities to the number of non-vulnerabilities (code smells and bugs). To separate the vulnerabilities, a filter is applied when parsing each tool's report. For PMD, violations are attributed with a specific ruleset, and vulnerabilities are only included within the "Security" ruleset. FindSecurityBugs also tags violations with a "SECURITY" category. SonarQube's REST API allows J-WAVE to query the

Figure 2: Proportions of security vulnerabilities vs. code smells or bugs identified by each SAST tool.

vulnerabilities without needing to examine other violations. Semgrep is built to only detect vulnerabilities. Finally, the most difficult report to parse is Yasca's. Each issue is tagged with a rule but not an overall category. Therefore, each rule found must be inspected to identify whether it constitutes a vulnerability within the given application.

4.1 WebGoat Findings

WebGoat is a project containing 348 files developed by the OWASP Foundation for testing SAST tools. The only tools which identified vulnerabilities in WebGoat were Semgrep and Yasca. PMD found zero vulnerabilities across all six projects scanned. SonarQube also did not report vulnerabilities, but multiple "security hotspots" were identified. These hotspots are warnings of potential vulnerabilities such as hard-coded secrets and URLs. These were filtered out of these results but may be worth investigating further. Finally, FindSecurityBugs did not report violations as it encountered multiple errors during the scan. The issue is that the underlying component, SpotBugs, requires that all external libraries be included within the scan directory. For standard scanning within an IDE, this is not an issue; however, J-WAVE does not have access to these

external libraries. Consequently, SpotBugs cannot find the required libraries and the scan fails. A potential solution is to include an auxiliary path in the run command, but this is difficult for projects compiled with build tools such as Maven or Gradle. J-WAVE currently does not support this approach, though it is a potential future enhancement.

Semgrep and Yasca were able to detect a multitude of vulnerabilities. Both found multiple cases of SQL information leaks, such as SQL commands in comments or non-prepared statements. Both tools also found cases of the insecure MD5 hash algorithm. Individually, Yasca identified instances of Java's `Process` class running commands with host permissions, weak credentials, and cases of unbuffered, arbitrary file reading that could lead to denial of service (DoS) or file disclosure attacks. Semgrep found cases of vulnerable functions within the Java Math library, object deserialization risks, and insecure HTTP transaction procedures. Although Semgrep rules were collected normally, certain Yasca violations were filtered out. Violations of rules "Poor Logging Practices" and "Cross Site Scripting" often coincided, as they are reported for all calls to Java's `System.out` functions. "External Links" and "Unique String" violations were similar to SonarQube's security hotspots in that their contents should be reviewed, but are not an inherent security risk. It should be noted that Yasca also detects violations within code inside comments and it reports each issue twice; as a result, the numbers reported in both figures were halved from the raw findings.

4.2 Findings from Data Structures and Algorithms

For the Data Structures and Algorithms course, 716, 969, 849, and 641 files were scanned for the four projects respectively. No vulnerabilities were found across the board for PMD, Semgrep, and SonarQube. Similar to WebGoat, SonarQube reported multiple security hotspots but no vulnerabilities.

From Figure 1, we see that a few vulnerabilities were identified by FindSecurityBugs and Yasca. All FindSecurityBugs reports concerned violations of reading or writing to arbitrary files. Yasca detected this problem in significantly more instances; for example, in the third project, FindSecurityBugs reported 6 cases of arbitrary file traversal while Yasca reported 120. Yasca also detected multiple potential DoS vulnerabilities. As with WebGoat, "Cross Site Scripting" and "Poor Logging Practices" violations were filtered out. "SQL Injection" and "SQL Information Leak" violations were also filtered out as they were flagged due to `System.out` calls or asterisks in Javadoc comments.

From these results, a variety of lessons can be taught on improving security. One example is a lesson on properly reading from or writing to specific files as opposed to those arbitrarily passed in. Additionally, buffers should be used when performing file I/O. Proper logging practices and conventional storage of strings are also reported and can provide useful insight to students and professors alike.

4.3 Findings from Web Application Development

While scanning the Web Application Development submissions, 9,454 files were passed as input. Once again, PMD and FindSecurityBugs (due to the library issue) identified no security

violations. Yasca findings of potential SQL Injection and DoS attacks were similar to those found within WebGoat. Semgrep and SonarQube identified the same series of vulnerabilities as one another, including cases of insecure hostname verification, insecure certificate verification, and insecure SSL protocols.

These results lead to two primary lessons. The first concerns protecting SQL commands through the use of prepared statements. The second enforces secure hostname and certificate verification practices along with using secure transfer protocols such as Transport Layer Security (TLS).

4.4 Results Summary

From our results, we observe that J-WAVE is capable of handling large volumes of input data properly. These stress tests are useful for educators planning on scanning student code in bulk as opposed to submission by submission. Therefore, educators can more easily generalize common vulnerabilities programmed by their students.

It is also clear that not every tool is equipped to handle all scenarios. For general coursework, it is more beneficial to use tools that check for code smells and bugs as opposed to vulnerabilities. Tools such as PMD and SonarQube are useful here as their reports provide information on how to best document, optimize, and design code. Yasca is another tool that can be useful for these projects to emphasize proper logging techniques.

The tools better equipped for vulnerability detection should be prioritized more often when observing web applications. Tools such as Semgrep and Yasca were useful in observing information regarding stored secrets, SQL injection, as well as insecure protocols and functions. Unfortunately, the reliance of external libraries makes it difficult to accurately test FindSecurityBugs' capabilities within J-WAVE. Very few files were scanned by FindSecurityBugs for the Web Application Development and WebGoat projects, so it is difficult to draw conclusions regarding this tool.

5 Lessons Learned

J-WAVE's ability to process thousands of input files is useful for course-wide scanning. Batch submissions can scan all student submissions in bulk. The stress test of scanning over 9,000 files for the Web Application Development course shows that this robustness useful for large classes.

The SAST tools used are complementary and should be used together to detect a wider range of problems. PMD is most useful for code smell and bug detection. SonarQube is similar to PMD in that it is best at identifying code smells and bugs, but provides more security rules. FindSecurityBugs was able to detect some issues within the Data Structures and Algorithms course projects, so it might be useful to pair alongside another tool for scanning general projects. Semgrep and Yasca are the best tools for security scans as they detect actual instances of vulnerabilities. That being said, Yasca's reports should be analyzed more thoroughly as it is prone to producing false positives.

PMD and SonarQube reports should be prioritized within general applications. Although vulnerabilities can be detected in general applications, the greater focus should be on code design issues. The Data Structures and Algorithms course projects are good examples where these tools should be prioritized. These projects do contain some potential vulnerabilities, but the code smell and bug violations are more frequent and identifying them provides greater potential value. The vulnerabilities found—proper logging practices, buffered input and output streams, and non-arbitrary file disclosure—are closer to code design issues than security flaws. In addition, focusing on the output of two tools as opposed to five is less overwhelming for both students and professors.

Semgrep and Yasca reports should be prioritized for scans of web applications. The majority of the detected vulnerabilities occurred within WebGoat and the Web Application Development project submissions. Semgrep consistently reports only vulnerabilities, and its capabilities can be further improved should individuals look towards Semgrep Pro. On the other hand, Yasca reports do contain multiple false positives, but with a deeper analysis the reports can be quite useful. In addition, Yasca picked up on certain vulnerabilities not found by Semgrep, so it is worth using to cover up flaws with Semgrep Community. Together, these tools detected a multitude of issues within the Web Application Development submissions. From these issues, we were able to come up with lessons on SQL statement protection, prepared statements, secure host-name and certification validation, and secure transfer protocols.

6 Problems Faced

During J-WAVE development, there were multiple problems that arose. To begin, when running Sonar scans through J-WAVE, they would arbitrarily fail when running on a Mac with an Apple M1 or M2 chip. The reason for these failures is that the container offered by Sonar to run these scans in is built upon the AMD64 platform that hosts a version of Qemu that often fails when run on an ARM64 platform. It is unclear whether Docker, Sonar, or Qemu are responsible for the fix, but development of J-WAVE was continued on Windows for this reason. The two additional containers for running SonarQube also prefer the AMD64 platform. Otherwise, the persisted volume data will not properly load, and thus all J-WAVE SonarQube scans will fail. Lastly, the container for Sonar scanning suffers from crashes when there exists large amounts of output. These crashes in testing occurred due to mass duplication of certain files, but they could potentially occur for other reasons as well.

Another issue was that J-WAVE depends the Java JDK version used. FindSecurityBugs and Semgrep have not been updated to work alongside the latest JDKs, including JDK 21. As a result, the newest version of the tomcat:10 Docker image cannot be used as the base for J-WAVE's environment; tomcat:10.0.0 with JDK 11 was used instead, but other tags may also work.

7 Conclusion

In conclusion, J-WAVE as a tool can be used to scan code with the goal to further vulnerability education. The SAST tools it embodies are complementary, and they should be used together to detect a wider range of problems. Combining these tools together, they are able to overcome the weaknesses of individual tools with a broader scope and more detection rules. However, it may be

better to focus on certain tools depending on the desired amount of output as well as the type of project being scanned. PMD and SonarQube reports should be prioritized within general applications as they are more optimal for bug or code smell detection. Semgrep and Yasca reports should be prioritized for scans of web applications. These tools are better for honing in on vulnerabilities, but they come with aforementioned limitations. FindSecurityBugs is difficult to evaluate in J-WAVE's current state, but it can be improved to better support the inclusion of external libraries.

J-WAVE is an all encompassing tool that offers critiques of all kinds to input code. With J-WAVE, students can analyze their own code for vulnerabilities to learn on their own time. Alternatively, J-WAVE's ability to take in thousands of input files is very useful for professors looking to scan their students' applications for violations. From those violations, they can develop lessons on how to proactively prevent, identify, and mitigate common vulnerabilities.

References

[1] "Browse CVE vulnerabilities by date," accessed: 20-Jan-2026. [Online]. Available: https://www.cvedetails.com/browse-by-date.php

[2] O. Foundation, "OWASP top ten," accessed: 20-Jan-2026. [Online]. Available: https://owasp.org/www-project-top-ten/

[3] M. Preston, "The differences between SCA, SAST and DAST," CloudDefense.ai Blog, Sep 2023, accessed: 20-Jan-2026. [Online]. Available: https://www.clouddefense.ai/blog/the-differences-between-sca-sast-and-dast

[4] M. Alenezi and Y. Yasir, "Developer companion: A framework to produce secure web applications," *International Journal of Computer Science and Information Security (IJCSIS)*, vol. 14, no. 7, Jul 2016.

[5] N. Sinhabahu, P. Wimalaratne, and C. Wijesiriwardana, "Secure codecity with evolution: Visualizing security vulnerability evolution of software systems," in *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2020, pp. 1–2.

[6] A. Sabatini, "Evaluating the testability of insecure deserialization vulnerabilities via static analysis," Master's thesis, University of Milan, Milan, 2021.

[7] NIST, "Source code security analyzers," Apr 2023, accessed: 20-Jan-2026. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers

[8] "PMD," Dec 2025, accessed: 20-Jan-2026. [Online]. Available: https://pmd.github.io/

[9] O. Foundation, "OWASP find security bugs," accessed: 20-Jan-2026. [Online]. Available: https://owasp.org/www-project-find-security-bugs/

[10] "SEMGREP," accessed: 20-Jan-2026. [Online]. Available: https://semgrep.dev/docs/

[11] M. Scovette and C. Carson, "Yasca," accessed: 20-Jan-2026. [Online]. Available: https://scovetta.github.io/yasca/

[12] Sonar, "SonarQube," accessed: 20-Jan-2026. [Online]. Available: https://www.sonarsource.com/products/sonarqube/