GEORGE BROWN PRESS
JOURNALS, BOOKS & PROCEEDINGS

Journal of Computer, Signal, and System Research

Article

Optimization of Vulnerability Detection and Repair Strategies Based on Static Application Security Testing

Shuang Yuan [1,*]

[1] Technology Risk Management, American Airlines, Fort Worth, Texas, 76155, United States
* Correspondence: Shuang Yuan, Technology Risk Management, American Airlines, Fort Worth, Texas, 76155, United States

Abstract: Static Application Security Testing (SAST), as an important vulnerability detection and repair technology, plays a key role in ensuring software security. However, existing static application security testing still faces challenges such as delays in vulnerability detection and inefficiencies in the repair process. This paper starts with the analysis of the current situation, discusses the problem of the delay of vulnerability detection and the inefficiency of repair, and puts forward the specific method of optimizing the vulnerability detection strategy and repair strategy of static application security testing. By introducing multi-dimensional analysis to improve detection accuracy, optimizing static analysis algorithm to improve detection efficiency, and applying automated and intelligent repair strategies, the purpose is to improve the efficiency and effect of vulnerability detection and repair. It is hoped that the implementation of optimization strategy can provide a more efficient solution for security protection in software development.

Keywords: static application security testing; vulnerability detection; bug repair; optimization strategy; automate

1. Introduction

In today's information society, software security issues are becoming increasingly important. Vulnerabilities in application programs often become entry points for attackers, bringing huge risks to enterprises and users. Static Application security testing (SAST), as an important means to detect and repair code vulnerabilities, has been widely used in the security of software development. By scanning source code, bytecode, or binary code, static analysis can detect potential security vulnerabilities early in development and avoid costly fixes later. However, static analysis tools still face problems such as delayed detection and low repair efficiency in practical applications, which affect the timeliness and quality of vulnerability repair. To address these challenges, this paper proposes a series of optimization strategies to improve the effectiveness and efficiency of static application security testing in vulnerability detection and repair.

2. Current Situation of Static Application Security Vulnerability Detection and Repair

2.1. The Lag of Vulnerability Detection

The lag of static application security testing (SAST) in vulnerability detection is mainly reflected in the timeliness and accuracy of analysis results. As the software development process accelerates and the source code is constantly updated, static analysis tools often cannot keep up with the changes, resulting in delayed detection results. This lag is mainly due to the scanning mechanism of static analysis tools [1]. The analysis process is complex and time-consuming, especially when dealing with large code bases, causing scan cycles to grow significantly. As a result, developers often face the problem that the

Received: 05 May 2025
Revised: 10 May 2025
Accepted: 29 May 2025
Published: 30 May 2025

detection results are inconsistent with the current code status when conducting vulnerability detection. In addition, static analysis tools are mainly based on the way of matching rules and patterns in vulnerability identification. This method is usually not flexible enough in the face of changing and complex program architecture and new vulnerability types, which may cause the omission or misdetection of vulnerabilities. Tools' detection rules are often based on the characteristics of known vulnerabilities, and static analysis is less adaptable to unknown or emerging vulnerabilities. The detection of vulnerabilities often lags behind their introduction, causing developers to be unable to identify and patch new security problems in a timely manner. In addition, the disconnect between static analysis tools and the development process is also a cause of lag problems. Existing static analysis tools are not effectively integrated with the development environment, resulting in analysis results not being fed back to developers in real time, which delays the speed of vulnerability detection [2].

2.2. The Inefficiency of Bug Fixes

In static application security testing (SAST), the inefficiency of vulnerability repair is mainly due to excessive manual intervention in the repair process, poor adaptability of the repair scheme, and insufficient follow-up verification. At present, the fix recommendations provided by most static analysis tools are usually static and basic, and do not implement intelligent automatic repair functions, resulting in the need for manual intervention in bug repair. While some tools provide automated repair capabilities, their repair recommendations are often not targeted and lack intelligent reasoning for complex vulnerability scenarios, resulting in a significant amount of manual adjustment. The manual repair mode not only prolongs the repair cycle but also easily produces new errors or omissions in the repair process, affecting the repair effect. The existing repair strategies lack sufficient flexibility and scalability. With the expansion of project scale or changes in the development environment, the repair schemes of static analysis tools often cannot quickly adapt to new requirements. This failure to respond timely to changing development needs leads to inefficiency in the repair process. Traditional restoration schemes are usually fixed and singular. They lack variety and adaptability, and cannot be flexibly adjusted according to actual situations, limiting their application in complex engineering projects. At the same time, the regression testing and verification after vulnerability repair are inefficient. Code that has been fixed often needs to undergo extensive regression testing and validation to ensure that the fix does not introduce new bugs. However, regression testing often lacks automation and efficiency [3]. The high cost and inefficiency of manual testing make the verification process extremely complex and time-consuming. Especially in large-scale projects, regression testing is often the most time-consuming part of the repair process. These problems together lead to the inefficiency of the vulnerability repair process, which greatly affects the timeliness and accuracy of the repair work.

3. Optimize Vulnerability Detection Strategies Based on Static Application Security Testing

3.1. Introduce Multidimensional Analysis to Reduce False Positives and False Positives

False positives and false negatives in static application security testing (SAST) often stem from single rule matching and lack of context analysis, which limits the accuracy of vulnerability detection. By combining data flow analysis, control flow analysis, semantic analysis, and other methods, the introduction of a multi-dimensional analysis strategy can identify potential vulnerabilities in a comprehensive manner and reduce false positives and false negatives. Through data flow analysis, we can track the flow of data within the program, grasp the whole process of variables from creation to destruction, and then find the risk points of data leakage or illegal operations. Control flow analysis focuses on reviewing the process execution path and identifying possible underreporting under complex conditions [4]. From the perspective of business logic and functional requirements,

semantic analysis evaluates whether the actual behavior of the program is reasonable and identifies potential security risks. In addition, the analysis combined with the context and historical data of the program helps to improve the accuracy of the detection. Through cross-validation of multi-dimensional data, false positives that may be caused by a single analysis method can be eliminated, and the efficiency and effect of vulnerability detection can be improved. In the analysis process, the probability of vulnerability detection $P_{det\,ection}$. And the probability of underreporting P_{miss}. They are:

$$P_{det\,ection} = \frac{TP}{TP+FP}, P_{miss} = \frac{FN}{FN+TN} \quad (1)$$

TP represent true cases (real vulnerabilities detected), FP false positive, FN false counter example, TN true counterexample. In addition, by introducing machine learning algorithms, the model can be trained to recognize common false positive patterns for dynamic adjustment. Machine learning not only continuously refines the rules during scanning, but also automatically adjusts the classifications and priorities based on the context of the vulnerability. For example, the algorithm can flexibly adjust the repair scheme according to the influence range of the vulnerability, the difficulty of repair and other factors, and effectively improve the accuracy and efficiency of detection. The multi-dimensional analysis strategy can effectively reduce false positives and false negatives in static analysis, thereby improving the accuracy of vulnerability detection, reducing unnecessary manual intervention, and improving the overall development efficiency and security.

3.2. Improve Collaboration between Static Analysis and the Development Tool Chain

Static analysis tools alone may not meet the needs of rapid development and continuous integration, so working in concert with the development tool chain is critical. Static analysis is tightly integrated with integrated development environments (ides), version control systems, and continuous integration (CI) systems, enabling real-time feedback and automated vulnerability detection. Integrating static analysis tools into the IDE makes it easier for developers to receive feedback on security vulnerabilities at the coding stage. This IDE integration enables developers to identify and fix potential security risks during the code writing and debugging phase, reducing the cost of later fixes. Combining static analysis with a version control system can automatically trigger vulnerability scanning when code is committed or merged to analyze new and modified code. The combination with CI system further improves the degree of automation of vulnerability detection. Whenever the code is updated, the CI system automatically runs a static analysis tool, scans the code base, and generates a report that gives feedback to the developer to fix it. Automated real-time feedback greatly improves the efficiency of vulnerability detection and reduces the time required to fix vulnerabilities during the development phase. As the DevOps culture advances, automated toolchains and continuous integration processes are being optimized to make vulnerability detection and remediation more efficient. This collaborative working mechanism can also play an advantage in large-scale team development, automation tools can centralize vulnerability information into a security issue tracking system, facilitate collaboration and problem tracking among team members, and constantly fix vulnerabilities through automated testing to maintain software security and stability [5].

3.3. Optimize Static Analysis Algorithm to Improve Scanning Efficiency

The optimization of static analysis algorithm is the key to improve the efficiency of vulnerability detection. Traditional static analysis methods are usually based on recursive search and exhaustive algorithms, which may lead to slow detection process and huge resource consumption as the project scale expands. In order to improve the scanning efficiency, it is necessary to optimize the algorithm design and architecture. Using incremental analysis technology can greatly improve the efficiency of scanning. This technique performs a local scan of the changed code by comparing the differences between the current

version and the previous version, avoiding the need to scan the entire project. This approach reduces unnecessary calculations, improves efficiency, and ensures the timeliness of vulnerability detection. With parallel processing and distributed analysis architectures, the speed of static analysis is significantly increased. By using distributed computing frameworks such as Apache Hadoop and Spark, analysis tasks can be distributed to multiple computing nodes, and multi-core and multi-machine resources can be used to process multiple analysis tasks in parallel, thus shortening the overall analysis time, which is especially suitable for processing large-scale complex projects. The total time of parallel computation can be expressed by the following formula:

$$T_{parallel} = \frac{T_{total}}{n} \quad (2)$$

T_{total} is the total time required for single-core processing, n indicates the number of nodes in parallel processing. The time complexity of optimization algorithm is also an important means to improve efficiency. By using a more efficient path analysis algorithm instead of the traditional algorithm, it can ensure the comprehensiveness of detection, reduce the computational complexity and reduce unnecessary calculations, thereby increasing scanning speed, accelerating vulnerability detection and remediation, and significantly enhancing development and maintenance efficiency.

4. Optimize Vulnerability Remediation Strategies Based on Static Application Security Testing

4.1. Improve the Automation and Intelligence of Vulnerability Repair

With the increasing complexity of software development, the automation and intelligence of vulnerability repair are particularly important. Automated repair can not only significantly improve repair efficiency, but also reduce the workload of developers. In traditional manual fixes, developers often rely on the results of static analysis to locate vulnerabilities and manually write fix code, which is not only inefficient, but also prone to errors. In contrast, automated repair tools can automatically provide repair suggestions or even implement repair directly through predefined repair rules and policies. For example, in the face of SQL injection vulnerabilities, such tools can automatically modify code to secure parameterized queries according to rules, avoiding manual omissions. Intelligent repair further improves the accuracy and intelligence level of vulnerability repair. By introducing a machine learning model, we analyze historical data to determine the most effective fixes and automatically generate fixes based on the specific circumstances of the vulnerability. Through numerous repair cases, these models can accurately identify and infer the most appropriate repair methods, reducing the need for manual intervention. When dealing with complex vulnerabilities, smart fixes provide developers with more precise solutions. Smart repair can also flexibly adjust repair strategies based on the context of the vulnerability, reduce the occurrence of errors, and improve overall repair quality. The combination of automation and intelligence makes the vulnerability repair process more efficient and accurate, enhances the security of the software, and ensures the quality and safety of the code during the development process. With the continuous development of artificial intelligence technology, intelligent repair can not only effectively detect vulnerabilities, but also predict possible vulnerabilities through big data analysis and automatically implement protective measures.

4.2. Improve the Scalability and Flexibility of the Repair Strategy

The scalability and flexibility of a bug fix strategy is a key factor in dealing with an increasingly complex and changing development environment. With the rise of new development methods such as microservice architecture and containerization technology, vulnerability repair is no longer the repair of a single code module, and more and more vulnerability repair needs to consider the impact of cross-module and cross-platform. Therefore, the scalability of the remediation strategy is particularly important, and it must be compatible with multiple development environments, architectures, and technology

stacks to ensure that the remediation measures can be effectively implemented in a variety of different environments. The extensibility of this strategy also requires that it work smoothly across a variety of development frameworks and toolchains. For large scale or microservices-based systems, the fix strategy should be able to coordinate effectively across multiple services, overcoming the limitations of a single codebase or technology stack. In addition, the remediation strategy should not only be modular, but should also be able to match multiple development tools (such as integrated development environments, continuous integration and continuous deployment toolchains, etc.) and various environments (such as cloud computing environments, containerized environments, etc.). In terms of flexibility, the fix strategy should automatically match the appropriate fix based on the type of vulnerability. For example, for some serious vulnerabilities, a system-level architectural tweak may be required, while for other, more minor vulnerabilities, a simple code fix may be sufficient. Therefore, in the solution to the vulnerability, the repair method should be flexibly adjusted according to the complexity and repair cost of the vulnerability, so as to avoid over-repair or missing important fixes. At the same time, the repair strategy should also have a certain personalization function, so developers can adjust repair priorities and workflows according to specific project requirements and security policies. In addition, the scalability of the fix strategy needs to take into account different team and business needs, allowing for rapid adaptation and updates. Especially in the case of the implementation of microservices architecture and containerization technology, the repair strategy must be flexible to respond to the requirements of different versions and different services, to ensure that the vulnerability repair can fully cover each component and platform, and to ensure the efficiency and comprehensiveness of the repair measures. By improving the scalability and flexibility of the repair strategy, we can ensure that a variety of complex vulnerability repair needs are addressed in different development environments, thereby improving overall security and development efficiency.

4.3. Strengthen Regression Testing and Verification after Vulnerability Repair

Regression testing and verification after vulnerability repair are critical to ensuring the effectiveness of fixes and preventing the introduction of new vulnerabilities. This test verifies not only that the vulnerability has been fixed, but also whether new issues have emerged during the repair. In order to guarantee the quality of repaired code, implement effective regression testing strategy. The core of regression testing is to fully scan the repaired application using an automated testing framework to ensure comprehensive fixes and validate that system functionality remains intact. Maximizing test coverage is a core objective of any regression testing strategy. The quality of regression tests can be expressed by the following formula:

$$Quality_{test} = \frac{C_{after} - C_{before}}{C_{before}} \times 100\% \quad (3)$$

The test coverage before vulnerability repair is C_{before}, the test coverage after repair is C_{after}, this formula shows that the quality of regression testing depends on the improved test coverage after repair. High-coverage regression testing is an effective way to catch potential new vulnerabilities, especially those that may have been overlooked or introduced during the fix process. In addition to regular functional regression testing, performance regression testing is also very important. In the process of vulnerability repair, especially when the core code is modified, the performance of the system may be affected. Therefore, the performance regression test can ensure that the repaired system still runs stably under high concurrency scenarios through stress testing, load testing and other methods. By enhancing regression test coverage and incorporating functional, performance, and security tests, the quality of fixes can be ensured while minimizing the risk of new issues, thereby improving security and stability throughout the software development lifecycle.

5. Conclusion

With the diversification of network security threats, Static application security testing (SAST) has become an indispensable tool for ensuring software security. However, current static analysis tools still face the problem of lag and inefficiency in the process of vulnerability detection and repair. By introducing multi-dimensional analysis, optimizing static analysis algorithms, and improving automated repair capabilities, the accuracy and efficiency of vulnerability detection can be significantly improved, and the repair process can be optimized. In the future, with the continuous development of artificial intelligence and automation technology, the vulnerability detection and repair strategy of static application security testing will be more intelligent and refined, providing a more efficient and reliable solution for security protection in the software development process. Further research and practice will also drive technological advances in this area to meet the growing security needs.

References

1. K. Gong, X. Song, N. Wang, C. Wang, and H. Zhu, "SCGformer: Smart contract vulnerability detection based on control flow graph and transformer," *IET Blockchain*, vol. 3, no. 4, pp. 213–221, 2023, doi: 10.1049/blc2.12046.
2. L. Zhang, Y. Li, R. Guo, G. Wang, J. Qiu, S. Su, et al., "A novel smart contract reentrancy vulnerability detection model based on BiGAS," *J. Signal Process. Syst.*, vol. 96, no. 3, pp. 215–237, 2024, doi: 10.1007/s11265-023-01859-7.
3. B. Xia, W. Liu, Q. He, F. Liu, J. Pang, R. Yang, et al., "Binary vulnerability similarity detection based on function parameter dependency," *Int. J. Semant. Web Inf. Syst.*, vol. 19, no. 1, pp. 1–16, 2023, doi: 10.4018/IJSWIS.322392.
4. S. Gephard, "Transplantation of prespawn adult Sea Lampreys as an important restoration strategy," *N. Am. J. Fish. Manage.*, vol. 43, no. 6, pp. 1584–1595, 2023, doi: 10.1002/nafm.10941.
5. B. Korkut, E. T. Bayraktar, D. Tağtekin, H. Çolak, and M. Özcan, "Cracked tooth syndrome and strategies for restoring," *Curr. Oral Health Rep.*, vol. 10, no. 4, pp. 212–222, 2023, doi: 10.1007/s40496-023-00352-1.