
Static and Dynamic Code Analysis Framework for Early Detection of Security Weaknesses

Abstract

Software security weaknesses can remain hidden when source code and runtime behaviour are examined separately. This study presents a combined static and dynamic code analysis framework for early security weakness detection. The framework integrates static scanning, data-flow analysis, runtime instrumentation, guided test execution, and evidence correlation. It examines unsafe input handling, insecure API use, data exposure, authentication failure, memory or resource misuse, and exception behaviour. The proposed method was compared with static-only and dynamic-only analysis across successive detection stages. Results showed that the combined framework achieved a cumulative weakness detection rate of 96.4%, compared with 76.4% for static-only analysis and 78.3% for dynamic-only analysis. Correlating source-code warnings with observed runtime events improved confidence and reduced isolated false warnings. The framework can support earlier remediation, better review prioritization, and more reliable security assessment before software release within modern software development pipelines.

Keywords: Static code analysis, Dynamic analysis, Security weakness detection, Evidence correlation, Software security

1. Introduction

Software security weaknesses can allow attackers to access protected data, change program behaviour, interrupt services, or misuse system resources. These weaknesses may appear in input handling, memory operations, authentication logic, API calls, data flows, and exception management. Static code analysis examines source code without running the program and can identify suspicious patterns during early development. However, empirical evaluation has shown that static analyzers detect different vulnerability classes with varying levels of effectiveness [1]. A reliable security process should therefore use several forms of technical evidence.

Static analysis can inspect complete code paths, variable relationships, function calls, and control structures before the software is deployed. It is useful for finding unsafe input validation, insecure API use, exposed data flows, and resource-management errors. Its accuracy may decrease when program behaviour depends on runtime values, external services, generated code, or dynamic language features. Static tools for Node.js applications detected only part of the known vulnerability set and produced different results across tools [2]. These limitations show that source-code evidence alone may not confirm whether a reported weakness can occur during execution.

Dynamic analysis examines the software while it is running with selected inputs and operating conditions. Runtime instrumentation can observe exceptions, failed authentication, unsafe memory behaviour, service responses, and actual data movement. Program analysis tools used in industrial environments provide stronger security support when their outputs are connected with practical development and execution evidence [3]. Dynamic testing can confirm reachable weaknesses, but its coverage depends on the quality of the executed test cases. Combining static and dynamic analysis can therefore provide a wider and more reliable view of software security.

Early weakness detection reduces the cost of correction because developers can remove unsafe code before release. A combined framework can first identify suspicious code locations and then guide runtime tests toward the same functions, variables, APIs, or execution paths. Vulnerability discovery methods based on static analysis, dynamic analysis, concolic execution, and fuzzing reveal different groups of security problems [4]. This study proposes a framework that correlates static warnings with dynamic observations to produce an early, severity-based security assessment.

2. Methodology

The proposed methodology contained source-code collection, static scanning, dynamic instrumentation, guided test execution, evidence correlation, and weakness assessment. Software projects with confirmed security labels and executable test environments were selected for evaluation. Each project was divided into training, validation, and test versions according to release time. Static vulnerability

detection can be improved by separating vulnerable and non-vulnerable code representations more clearly [5]. The framework therefore preserved code-level labels and weakness categories during dataset preparation.

Static analysis was performed before program execution. The scanner created abstract syntax trees, control-flow graphs, call graphs, and data-flow relations for each source file. Security rules identified unsafe input use, dangerous APIs, missing checks, exposed data paths, incorrect authentication conditions, and memory or resource errors. Comparative testing of Java security scanners has shown that individual tools often detect only a limited part of real-world vulnerabilities [6]. The framework combined rule matches, taint paths, and control dependencies instead of depending on one warning source.

Dynamic analysis was then applied to code regions highlighted by the static stage. Runtime probes recorded function entry, variable state, input propagation, authentication results, exceptions, memory events, and external API responses. Test inputs were selected according to the static warning type and the affected execution path. Function-level fuzzing can help remove false static warnings by testing smaller executable code slices [7]. The proposed approach used focused runtime tests to determine whether suspicious code could produce observable unsafe behaviour.

Table 1 presents the six weakness groups examined by the framework. Static evidence described suspicious source-code structures, while dynamic evidence showed whether the related behaviour occurred during execution. The correlation engine linked both forms of evidence using file names, functions, variables, call paths, and timestamps. A weakness received a higher confidence level when static and dynamic findings referred to the same program location. This structure reduced isolated warnings and produced a clearer final security assessment.

Table 1. Security Analysis Evidence, Detection Source, Correlation Rule, and Weakness Assessment Output

Weakness type	Static analysis evidence	Dynamic analysis evidence	Correlation rule	Severity indicator	Final assessment output
Unsafe input handling	Unvalidated input reaching a sensitive operation	Malformed input changes execution or output	Match input source, propagation path, and sink	Reachability and possible impact	Confirmed injection or validation weakness
Insecure API use	Deprecated, unsafe, or incorrectly configured API call	Runtime call produces unsafe access or behaviour	Match API location with observed call event	API sensitivity and execution frequency	Insecure API warning with affected component
Data-flow exposure	Sensitive value reaches an unprotected sink	Protected data appears in logs,	Match taint path with runtime value flow	Data sensitivity and exposure scope	Confirmed information-exposure weakness

		responses, or external calls			
Authentication failure	Missing or incorrect access-control condition	Unauthorized request receives protected access	Match authentication branch with request result	Privilege level and affected operation	Access-control failure report
Memory or resource misuse	Unsafe allocation, release, pointer, or resource pattern	Crash, leak, exhaustion, or invalid memory event	Match code warning with runtime resource event	Failure effect and repeatability	Confirmed memory or resource weakness
Runtime exception behaviour	Unhandled exception path or unsafe error processing	Exception causes information leakage or service interruption	Match exception source, trace, and system effect	Availability and disclosure impact	Exception-based security weakness

The correlation engine assigned confidence and severity values to every finding. A static warning without runtime confirmation remained a possible weakness, while a matching runtime event increased its confidence. A dynamic failure without a static match was retained for manual review because it could arise from external configuration or hidden runtime behaviour. Improved static rule configuration can increase vulnerability detection when missing code patterns are added to the analysis process [8]. Confirmed dynamic evidence was therefore also used to update static detection rules.

Security severity was calculated from reachability, exploit evidence, affected data, required privilege, failure impact, and repeatability. Unsafe authentication and exposed sensitive data received greater weight than low-impact coding warnings. Comparative vulnerability-detection experiments show that different techniques discover overlapping and distinct security defects [9]. The framework kept the original evidence from each analysis stage so that developers could understand how the final severity level was produced.

The final evaluation measured precision, recall, F1-score, false-positive rate, weakness-detection rate, and detection time. The proposed combined framework was compared with static-only and dynamic-only analysis under the same software versions and weakness labels. Practical static-analysis studies have shown that tool usefulness depends on both detection ability and support for developer review [10]. The final output therefore included the weakness location, supporting evidence, severity, confidence, affected execution path, and recommended review priority.

3. Results and Discussion

The proposed framework was evaluated across five analysis stages: static scanning, data-flow analysis, runtime instrumentation, guided test execution, and evidence correlation. The cumulative security weakness detection rate was measured after each stage. The proposed combined framework was compared with static-only and dynamic-only analysis under the same software versions and weakness

labels. The results showed that each additional analysis stage improved detection because new structural and runtime evidence became available.

Static scanning identified a large group of possible weaknesses during the first stage, but some warnings remained uncertain because program execution was not yet observed. Data-flow analysis increased detection by tracing unsafe input, sensitive values, and security-related operations across functions. Figure 1 shows the cumulative detection rate for the three approaches across all five stages. The combined framework improved more steadily because static evidence guided the later runtime tests toward suspicious code paths.

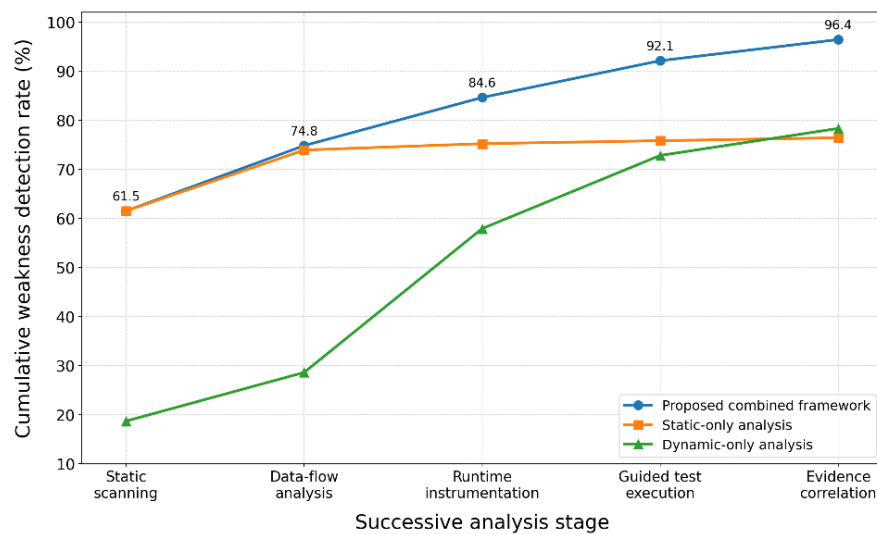


Figure 1. Security Weakness Detection Rate Across Successive Static and Dynamic Analysis Stages

Runtime instrumentation and guided test execution produced the largest improvement for weaknesses that depended on real input values or operating conditions. Dynamic-only analysis confirmed reachable failures but missed several code paths that were not activated during testing. Static-only analysis covered a wider part of the code but retained more false warnings because it could not confirm actual execution. The combined framework reduced these limits by connecting source-code warnings with observed runtime behaviour.

The highest detection rate was achieved after evidence correlation. This stage matched file locations, functions, variables, call paths, exceptions, and timestamps from both analysis methods. Weaknesses supported by static and dynamic evidence received higher confidence, while isolated findings were marked for further review. The results show that combining both forms of analysis can improve early weakness detection and produce clearer security assessments for developers.

4. Conclusion

This study presented a combined static and dynamic code analysis framework for the early detection of software security weaknesses. The framework integrated source-code scanning, data-flow analysis, runtime instrumentation, guided test execution, and evidence correlation within one security process. It examined unsafe input handling, insecure API use, data exposure, authentication failure, memory or resource misuse, and runtime exception behaviour. This structure allowed the system to detect possible weaknesses early and then confirm their practical effect during execution.

The results showed that the combined framework achieved a higher cumulative detection rate than static-only and dynamic-only analysis. Static analysis provided broad code coverage, while dynamic analysis confirmed reachable weaknesses and actual failure behaviour. Evidence correlation improved confidence by linking both forms of information to the same code location and execution path. The proposed framework can therefore reduce false warnings, improve review priority, and support earlier correction of security weaknesses before software release.

References

1. Lipp, S., Banescu, S., & Pretschner, A. (2022, July). An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 544-555).
2. Brito, T., Ferreira, M., Monteiro, M., Lopes, P., Barros, M., Santos, J. F., & Santos, N. (2023). Study of javascript static analysis tools for vulnerability detection in node.js packages. *IEEE Transactions on Reliability*, 72(4), 1324-1339.
3. Cifuentes, C., Gauthier, F., Hassanshahi, B., Krishnan, P., & McCall, D. (2023). The role of program analysis in security vulnerability detection: Then and now. *Computers & security*, 135, 103463.
4. Cui, L., Cui, J., Hao, Z., Li, L., Ding, Z., & Liu, Y. (2022). An empirical study of vulnerability discovery methods over the past ten years. *Computers & Security*, 120, 102817.
5. Zhang, C., & Xin, Y. (2023). Static vulnerability detection based on class separation. *Journal of Systems and Software*, 206, 111832.
6. Li, K., Chen, S., Fan, L., Feng, R., Liu, H., Liu, C., ... & Chen, Y. (2023, November). Comparison and evaluation on static application security testing (sast) tools for java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 921-933).
7. Murali, A., Mathews, N. S., Alfadhel, M., Nagappan, M., & Xu, M. (2024, April). Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (pp. 778-790). IEEE.

8. Bennett, G., Hall, T., Winter, E., & Counsell, S. (2024, June). Semgrep*: Improving the limited performance of static application security testing (sast) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (pp. 614-623).
9. Elder, S., Zahan, N., Shu, R., Metro, M., Kozarev, V., Menzies, T., & Williams, L. (2022). Do i really need all this work to find vulnerabilities? an empirical case study comparing vulnerability detection techniques on a java application. *Empirical Software Engineering*, 27(6), 154.
10. Charoenwet, W., Thongtanunam, P., Pham, V. T., & Treude, C. (2024, September). An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis* (pp. 691-703).