An overview of static and dynamic analysis in application security testing

Nguyen Thanh Cong[1], Le Huy Toan[2], Ta Minh Thanh[1*]

[1]Le Quy Don Technical University, 236 Hoang Quoc Viet, Bac Tu Liem, Hanoi, Vietnam;
[2]Center for Information Technology Product Testing – Department of Digital Transformation and Environment Resources Data Information – Ministry of Natural Resources and Environment, 28 Pham Van Dong, Cau Giay, Hanoi, Vietnam.
[*]Corresponding author: thanhtm@lqdtu.edu.vn
Received: 11 Aug. 2024; Revised 24 Oct. 2024; Accepted 12 Nov. 2024; Published 25 Nov. 2024.
DOI: https://doi.org/10.54939/1859-1043.j.mst.99.2024.1-11

ABSTRACT

In the context of increasingly complex information systems facing numerous cybersecurity threats, the evaluation of information security has become crucial. This paper focuses on two common methods of information security assessment: static analysis and dynamic analysis. Static analysis examines source code or binary code to detect security vulnerabilities during the software development phase. Dynamic analysis tests system security during operation, helping to identify vulnerabilities at runtime. The paper provides an overview of the techniques and tools for both methods, while comparing their advantages and disadvantages. Static analysis helps detect errors early but may miss runtime errors, while dynamic analysis performs real-world testing but can disrupt system operations. The combination of both methods yields the best results in ensuring information security.

Keywords: Information security; Static analysis; Dynamic analysis; Security vulnerabilities; Software testing.

1. INTRODUCTION

In the current digital age, information plays a crucial role in all fields, from business, education to healthcare and government. Information systems are becoming increasingly complex and integrating many new technologies, while simultaneously facing numerous security threats from cyberattacks, malware, and unauthorized intrusions. As a result, information security has become a key factor in ensuring the confidentiality, integrity, and availability of data and information systems. In this context, organizations need to ensure that their systems are protected as effectively as possible to prevent the exposure of sensitive information and data loss. Therefore, information security assessment has become an indispensable part of managing and operating information systems. Evaluating information security to detect and address security vulnerabilities before they can be exploited by attackers is essential. The two main methods used for information security assessment are static analysis and dynamic analysis. In this paper, we focus on studying two information security assessment methods: static analysis and dynamic analysis. Static analysis examines source code and binary code to detect security vulnerabilities from the software development stage. Dynamic analysis checks the security of the system during operation, helping to detect vulnerabilities at runtime. This research provides developers and information system administrators with an overview of these two methods, thereby helping them make appropriate choices to ensure information security for their systems. We have researched, evaluated, and compared the two methods of static analysis and dynamic analysis to propose a comprehensive solution for ensuring information security. Each method has its own advantages and limitations, suitable for specific stages and objectives in the process of system development and operation. Therefore, combining both

methods will yield the best results, helping to detect and address security vulnerabilities comprehensively and timely. The remainder of this paper is organized as follows: Section 2 presents the theoretical foundations of static and dynamic analysis. This section explores various techniques employed in both static and dynamic analysis, including code similarity-based and code pattern-based vulnerability detection for static analysis, and penetration testing and fuzzing for dynamic analysis. It also surveys current static analysis tools and discusses their selection criteria. Section 3 offers a comparative analysis between static and dynamic assessment methods, highlighting their respective strengths and limitations. This section also discusses the method of integrating static and dynamic analysis, emphasizing the benefits of combining both approaches for comprehensive vulnerability detection. Additionally, it addresses the challenges in assessing the risk level of security vulnerabilities, including common risk assessment methods like CVSS and CWE, and the difficulties faced in accurately identifying and evaluating vulnerabilities. Finally, section 4 concludes the paper by summarizing the key findings and discussing the importance of integrating both static and dynamic analysis methods in ensuring comprehensive information security. This section emphasizes the complementary nature of these approaches and their collective role in providing a holistic view of system security. It also suggests potential directions for future research in this field, particularly in improving the integration of static and dynamic analysis techniques and addressing the challenges in vulnerability risk assessment.

2. THEORETICAL FOUNDATIONS

In this section, we present static and dynamic analysis, along with several techniques used in both approaches. Additionally, we survey some current static analysis tools.

2.1. Static Analysis

Static analysis is a software examination technique that focuses on thoroughly reviewing the source code, binary code, or design documents of a system without executing the program. The primary objective of static analysis is to early identify security vulnerabilities, logical errors, and other potential issues during the software development process. This enables developers to proactively detect and address these issues before they can be exploited by attackers. Currently, traditional vulnerability analysis methods often require security experts to possess extensive specialized knowledge and practical experience. This limits the application of these methods in projects and reduces their effectiveness. At present, the application of deep learning and natural language processing technologies can intelligently process information about security vulnerabilities to support vulnerability research and enhance the efficiency of vulnerability exploitation. Binary-based vulnerability detection methods offer high accuracy and broad deployment capabilities but face challenges in tracking structural information and higher-level code information of the program. Currently, binary vulnerability detection employs reverse engineering methods to decompose binary code into a set of assembly codes, which are then used for information extraction and fed into artificial intelligence networks for training. Vulnerability detection techniques based on static analysis are divided into two categories: code similarity-based vulnerability detection and code pattern-based vulnerability detection [1]. The vulnerability detection techniques based on static analysis are categorized into two main types: code similarity-based vulnerability detection and

code pattern-based vulnerability detection. The differences between these two techniques are illustrated in table 1.

Table 1. *Comparison Between Code Similarity-Based And Code Pattern-Based Detection.*

Characteristics	**Code Similarity-Based Detection**	**Code Pattern-Based Detection**
Operating Principle	Searching for similar code segments	Identifying specific code patterns
Techniques Used	String comparison, syntax semantic analysis, machine learning, deep learning	Machine learning, deep learning
Advantages	Less dependent on data	Detection of new vulnerabilities
Disadvantages	Unable to detect new vulnerabilities	Dependent on data

2.1.1. Code Similarity-based Vulnerability Detection

Code similarity-based vulnerability detection is a technique used in cybersecurity to identify potential security vulnerabilities in software. The core idea of this method is that similar code segments (code clones) are likely to contain similar vulnerabilities [1]. These code similarity-based vulnerabilities are particularly exploited in web-based applications due to the frequent use of application services by many users. Four types of code clones are considered: Type-1 (T1): Identical code fragments, possibly differing in whitespace, layout, or comments. Type-2 (T2): Syntactically identical fragments with variations in identifiers, literals, types, whitespace, layout, and comments. Type-3 (T3): Copied fragments with modifications such as changed, added, or removed statements compared to Type-2. Type-4 (T4): Code fragments that perform the same computation but are implemented by different syntactic variants. The methods for detecting vulnerabilities based on code similarity can be divided into two main categories: Grammar-based detection and Semantic-based detection [1].

a. Grammar-based vulnerability detection

Grammar-based vulnerability detection is a static analysis technique used to search for potential security vulnerabilities in software source code. This technique focuses on analyzing the grammatical structure of the code, i.e., how code elements (such as variables, functions, statements, etc.) are combined to form the program. In the process of detecting similar code, the method of representing source code determines the limit of information extraction, which restricts the design of detection models and the selection of algorithms, ultimately affecting detection efficiency. Representing code at the syntactic level mainly considers the syntactic rules of the program's source code. The traditional text-based representation method, data processing only involves removing comments and spaces in the program code, mainly relying on text similarity measurement methods to detect copies at the text level [1]. Essentially, the implementation steps of this process are as follows:

1) Source code representation: This technique typically uses an Abstract Syntax Tree (AST) to represent the grammatical structure of the source code. An AST is a tree-

like representation where each node represents a code element and the edges represent the relationships between them.

2) AST analysis: Static analysis tools traverse the AST to search for code patterns that may cause security vulnerabilities. These patterns can be specific syntactic structures, unsafe uses of functions or variables, or combinations of these elements.
3) Comparison with known vulnerabilities: The detected code patterns are compared with a database of known security vulnerabilities. If a code pattern matches a known vulnerability, the tool will alert about the potential existence of a security vulnerability in the source code.

Additionally, applying artificial intelligence models is a very effective solution for detecting these types of vulnerabilities. Below is a comparison table of some studies using artificial intelligence models.

Table 2. *Comparison of several studies based on grammar-based detection methods.*

System/ Author	Data Preprocessing	Code Represe-ntation	Neural Network	Clone Type	Classification Object
CCLearner [2]	Uses ANTLR and ASRParser to parse each method, extract tokens C1-C3, and uses Eclipse ASTParser to create Abstract Syntax Tree (AST) for each method to extract additional tokens C4-C8.	Token, AST	DNN	T1-T3	Method pair
CLDH [3]	Analyzes each code segment into AST.	AST	LSTM	T1-T4	Code segment
White et al [4]	Uses ANTLR to encode source code.	AST	RtvNN	T1-T4	Method/file
Marastoni et al [5]	Uses Tigress C to obfuscate the dataset.	Binary image	CNN	T4	File

A notable advantage of syntax-based analysis is the ability to detect vulnerabilities related to the grammatical structure of the source code, even when the code segments are not completely identical. Furthermore, this technique can also detect new or unknown vulnerabilities by searching for unsafe code patterns. However, syntax-based analysis also has certain limitations. This process is more complex compared to text-based methods and requires the implementer to have a thorough understanding of the grammar of the programming language used. Additionally, another disadvantage is that this technique may miss vulnerabilities not directly related to the grammatical structure of the code.

b. Semantic-based vulnerability detection

Semantic-based vulnerability detection is a static analysis technique that focuses on examining the meaning of the source code rather than just relying on its syntactic structure. This method attempts to understand the control flow and data flow within the

program to determine whether code segments perform similar functions, thereby detecting potential vulnerabilities.

The operation of this technique includes the following steps:

1) Representation of control flow and data flow: These techniques use representations such as Data Flow Graph (DFG), Control Flow Graph (CFG), and Program Dependence Graph (PDG) to represent how data moves and how statements are executed.
2) Semantic analysis: Static analysis tools will analyze these graphs to understand the meaning of the source code.
3) Semantic comparison: Code segments with similar semantics (e.g., performing the same function) will be considered likely to contain similar vulnerabilities.

An example of semantic-based analysis: Suppose there are two code segments, one using a for loop and another using a while loop, but both perform the function of calculating the sum of numbers from 1 to 10. Semantic analysis can recognize that these two code segments have the same meaning, and if one code segment has a vulnerability, the other may also have a similar vulnerability. A notable advantage of the semantic-based vulnerability detection method is the ability to identify code segments that have equivalent functions, even when they have different syntactic structures. This allows this method to detect potential vulnerabilities that syntax-based methods might overlook. Moreover, semantic analysis provides a deeper insight into how the program operates and how data is processed, helping developers and security experts identify and patch security vulnerabilities more effectively. However, this method also has certain limitations. Semantic analysis is often more complex than syntactic analysis, requiring more sophisticated analysis techniques and tools. This process can also be more time-consuming and computationally intensive compared to syntactic analysis. Furthermore, creating semantic representations for different programming languages can be challenging. Some studies have focused on developing models based on this method, including:

Table 3. *Comparison of several studies based on semantic-based detection methods.*

System/Author	Data Preprocessing	Represen-tation	Neural Network	Clone Type	Classification Object
CCDLC [6]	Remove whitespace, normalize method blocks.	BDG, PDG, AST	CNN	T3-T4	Method block
Sheneamer et al [7]	Remove whitespace, normalize method blocks.	AST, PDG, BDG	-	T1-T4	Method block
ZEEK [8]	Divide procedures into basic blocks.	Hashes	NN	T4	Code block
DeepSim [9]	Use WALA to analyze bytecode.	DFG, CFG	DNN	T4	Method level

2.1.2. Code Pattern-Based vulnerability detection

Code Pattern-Based vulnerability detection is a technique for identifying security vulnerabilities that focuses on recognizing specific code patterns potentially containing vulnerabilities. This technique is typically divided into two phases: the training phase and

the detection phase. In the training phase, the following steps are performed:

1) Feature extraction: Utilize control flow and data flow analysis techniques to extract critical code segments (code gadgets) from known vulnerable programs.
2) Vector representation: Convert these critical code segments into feature vectors using techniques such as word2vec.
3) Model training: Use these feature vectors to train a deep learning model (typically a Convolutional Neural Network - CNN, Recurrent Neural Network - RNN, or Long Short-Term Memory network - LSTM) to recognize code patterns that are likely to contain vulnerabilities.

In the detection phase, the following steps are carried out:

1) Preprocessing: Perform preprocessing steps similar to those in the training phase on the new source code to be examined.
2) Utilizing the trained model: Use the trained deep learning model to predict whether code segments in the new source code contain vulnerabilities.

Several studies have employed the code pattern-based vulnerability detection method. VulDeePecker uses code gadgets and word embedding techniques to represent them, combined with a BLSTM neural network for model training. SySeVR also utilizes code gadgets but employs a custom word embedding algorithm and trains the model on six different deep neural networks. Meanwhile, CPGVA uses a code property graph to represent source code and combines word2vec with both CNN and RNN networks. Finally, Lee et al. focus on functions and use instruction2vec for word embedding of assembly code, then train the model using a Text-CNN network. All these studies use various datasets such as NVD, SARD, and Juliet to evaluate the effectiveness of their models. However, these methods still have some limitations, such as model training often relying on source code (which is not always available), the granularity of detection, and limited models. Additionally, a study by [10] comparing three vulnerability detection methods shows that these approaches have not adequately addressed the issues of cross-project and class imbalance in software vulnerability detection.

2.2. Dynamic Analysis

2.2.1. Penetration Testing

Penetration testing, or pentesting for short, is a method of testing network or system security by simulating cyberattacks from a hypothetical attacker. In this process, a team of security experts called "pentesters" or "red team" performs actions similar to those of a real attacker to identify security vulnerabilities in the system [11]. The goal of pentesting is to detect and exploit security vulnerabilities to improve the system's security level. Pentesters will attempt to find weaknesses and vulnerabilities in the system's structure, thereby assessing its reliability and safety in the face of potential external threats. Currently, some research focuses on automating the penetration testing process. There are two main approaches to automating penetration testing, including: Planning-based methods and Reinforcement Learning-based methods.

a. Planning-based methods

Planning-based methods are one of the earliest approaches to automate penetration testing (pentest). Instead of relying on complex machine learning techniques, these

methods use classical planning algorithms to identify potential attack sequences. Their strength lies in the ability to leverage existing models and knowledge about the system to make effective attack decisions. Amos-Binks et al. (2017) [12] proposed a new method for identifying attack plans, using automated planning to capture network attacks. The study demonstrated that the new approach to identifying attack plans, using automated planning, is effective in identifying network attacks. The metrics used in the study, such as Percentage Complete (PC) and Minimum Remaining Path Length (MRPL), have proven capable of tracking the progress of an attack. As the attacker gets closer to the target, these metrics will reflect this, providing network administrators with detailed information about the severity of the attack. Although the Chokepoint (CP) metric showed limited value in the experiments due to the linear nature of the attack graph considered, it still demonstrated accuracy in identifying critical attack steps. Overall, the study demonstrated the potential of using automated planning techniques to address basic network security issues. However, the use of the planning method still has some limitations, including adaptability to new environments and scalability, but it has proven useful in computational efficiency and interpretability in cases with small attack graphs.

b. Reinforcement Learning-based methods

Reinforcement Learning (RL) has been and is being widely applied in the field of automated penetration testing (pentest). Initial studies used POMDP (Partially Observable Markov Decision Process) to model the pentest process, aiming to capture the uncertain nature of the network environment. However, due to high computational complexity, these methods were only applicable to small-scale networks. Some studies have improved by decomposing and approximating the network, reducing complexity but ignoring the environment's changeability. To address this limitation, later studies switched to using MDP (Markov Decision Process), considering the result of actions as deterministic. This helps reduce computational complexity and focuses on the reality of actions, environment, and the presence of opponents. However, the assumption about the deterministic nature of actions may not fully reflect reality. Studies have used various RL algorithms such as Q-learning, Deep Q-learning (DQN), A2C, SARSA, PPO,... to train attack agents. These agents learn how to explore the network environment, exploit vulnerabilities, and find paths to the target. Some studies have combined graph embedding techniques to represent network states, helping to reduce the action space and improve algorithm performance [13]. Although reinforcement learning promises to bring many improvements to the field of automated pentesting, there are still challenges that need to be addressed. The scalability of current reinforcement learning algorithms is still limited, especially when applied to large-scale and highly complex networks. Additionally, the accuracy of reinforcement learning models also needs to be improved so that they can operate more effectively in real-world environments, where uncertain factors and unpredictable opponent behaviors frequently appear. Improving this accuracy will ensure that reinforcement learning models not only perform well in ideal environments but can also adapt and cope with complex situations in reality. Nevertheless, reinforcement learning still shows great potential in automating the penetration testing process, helping to improve the efficiency and accuracy of network security assessment.

2.2.2. Fuzzing

Fuzzing is an automated software testing technique that provides invalid, random, or unexpected input data to the application.

The goal of fuzzing is to detect unsafe data handling errors or other security vulnerabilities in the application [14]. There are many ways to classify fuzzing, but there are typically three main classifications [15], including: Generation-based and mutation-based fuzzing, "Dumb" and "Smart" fuzzing, White-box, Grey-box, and Black-box fuzzing. Based on input generation method, fuzzing can be generation-based or mutation-based. Generation-based fuzzing creates inputs from scratch based on the program's input specifications, while mutation-based fuzzing creates new inputs by modifying existing inputs. Based on the level of understanding of the target program, fuzzing can be "dumb" or "smart". "Dumb" fuzzing doesn't understand the input format and generates inputs through random mutations, while "smart" fuzzing leverages input models to generate more valid inputs. Based on the information used to guide the fuzzing process, fuzzing can be white-box, black-box, or grey-box. White-box fuzzing has access to the program's source code and uses program analysis to guide fuzzing, while black-box fuzzing treats the program as a "black box" and has no information about the program. Grey-box fuzzing uses lightweight instrumentation to obtain program information used to guide fuzzing. Each type of fuzzing has its own advantages and disadvantages. For example, white-box fuzzing can detect deeper errors in the program but requires access to the source code, while black-box fuzzing can be applied to any program but can only detect errors on the surface of the program. Grey-box fuzzing is a compromise between these two types; it uses program information to guide fuzzing but does not require access to the source code.

3. COMPARISON BETWEEN STATIC AND DYNAMIC ANALYSIS

3.1. Static and Dynamic Analysis

Static analysis focuses on analyzing source code or binary code without executing the program. The advantage of this method is its ability to detect vulnerabilities early in the development stage, without the need to execute the application, thus saving time and resources [16]. Static analysis also does not disrupt system operations. However, its disadvantage is that it cannot detect errors that only appear when the program is running and may produce many false results due to inaccurate environmental models [17]. Dynamic analysis, on the other hand, collects and analyzes data while the program is running. This allows dynamic analysis to detect runtime vulnerabilities and test the effectiveness of security measures in real-world environments [18]. However, this method can disrupt system operations and requires high expertise as well as a complex testing environment. Although static and dynamic analysis each have their own advantages and limitations, combining both methods can lead to higher effectiveness in detecting security vulnerabilities and improving application performance. Integrating static and dynamic analysis allows for leveraging the strengths of both approaches: static analysis helps identify potential errors in the source code without executing the program, while dynamic analysis can detect issues that only appear during execution. This is particularly important for modern web applications, where dynamic nature and complex interactions make individual analysis methods less effective. The combination of static and dynamic analysis is similar to white box penetration testing. The process of combining static and dynamic analysis manually can be described as follows [19]:

1) Source Code Review: Begin with a thorough static analysis of the application's source code. This step allows for identifying potential vulnerabilities and logical flaws that may not be apparent from external testing.
2) Vulnerability Mapping: Based on the source code review, create a map of potential vulnerabilities and areas of concern within the application.
3) Dynamic Testing Setup: Prepare a testing environment that mirrors the production setup as closely as possible.
4) Targeted Dynamic Analysis: Using insights from the static analysis, perform focused dynamic testing on the identified areas of concern. This may involve crafting specific inputs or test cases to trigger potential vulnerabilities.
5) Exploit Chain Development: Attempt to chain together multiple smaller vulnerabilities discovered during static and dynamic analysis to create more significant security issues.
6) Documentation and Reporting: Document the vulnerabilities found, including the path to discovery and potential impact. This comprehensive approach allows for demonstrating both the vulnerability and the thought process behind its discovery.

Furthermore, some research has focused on integrating static and dynamic analysis into a single automated process for vulnerability detection. In a study on phishing website detection [20], the authors created a tool that combines static analysis to extract features from HTML code with dynamic analysis by rendering the web page in a virtual browser. Results showed that this combined approach reduced the number of difficult-to-classify samples by 16%, significantly improving the accuracy of phishing website detection. Similarly, research by Di Lucca and Di Penta [21] proposed integrating two tools, WARE (static analysis) and WANDA (dynamic analysis), to enhance understanding and maintenance of web applications. WARE is used to generate Page Control Flow Graphs (PCFG) and identify Linear Independent Paths (LIPs), while WANDA records execution paths to identify LIPs covered during client-side dynamic page generation. These studies reveal a common process when combining static and dynamic analysis: (1) use static analysis to build a structural model of the application, (2) apply dynamic analysis to collect execution data, and (3) integrate results from both methods to provide a more comprehensive analysis. This process not only improves accuracy in detecting security vulnerabilities but also enhances understanding of complex web application behavior, especially in the context of increasingly dynamic and highly interactive web applications.

3.2. Assessing the risk level of security vulnerabilities

3.2.1. Common risk assessment methods

The two most common methods for assessing risk levels are CVSS (Common Vulnerability Scoring System) and CWE (Common Weakness Enumeration):

- CVSS: This system provides a standardized way to evaluate the severity of security vulnerabilities. CVSS uses a scale from 0 to 10, where 10 is the most severe. The score is calculated based on various factors such as attack vector, exploitation complexity, and impact on the system's confidentiality, integrity, and availability.
- CWE: This is a standardized list of software vulnerability and weakness types. CWE does not provide specific scores like CVSS, but it categorizes and describes in detail the types of vulnerabilities, helping developers and security experts better

understand the nature of the vulnerabilities.

3.2.2. Difficulties and Challenges

Accurately identifying the type of vulnerability and assessing it based on CVSS and CWE faces many challenges:

- Diversity of vulnerabilities: There are many different types of vulnerabilities, and each type can have multiple variants, making accurate classification difficult.
- Accuracy of tools: Static analysis tools often have a high false positive rate, especially when applied to large and complex programs
- Contextual assessment: CVSS and CWE require consideration of many contextual factors (e.g., deployment environment, impact) that automated tools find difficult to comprehensively evaluate.
- Continuous updates: New vulnerabilities constantly emerge, requiring frequent updates to databases and detection methods.

To overcome these challenges, a combination of automated methods and manual expert assessment is needed, as well as continuous updating and improvement of analysis tools and methods.

4. CONCLUSIONS

Through the analysis of static and dynamic analysis methods in information security, we have seen that both methods play crucial roles in the process of ensuring information security for a system. Static analysis focuses on examining and analyzing source code, binary code, or design documents without executing the program, while dynamic analysis performs tests in real or simulated environments by executing the program or using the system and observing its behavior. The combination of static and dynamic analysis provides a comprehensive view of the system's security level. Static analysis helps detect vulnerabilities from the early stages of software development and helps prevent issues before they are deployed, while dynamic analysis tests vulnerabilities under actual operating conditions of the system and helps ensure that implemented security measures are effective. In total, combining these two methods helps organizations gain a comprehensive view of their system's security level and helps them apply effective security measures to protect critical information and prevent potential network threats.

Acknowledgement: *This research is within the framework of the project "Research on developing technical regulations on testing and evaluating information security for web-based applications of Sector of Natural Resources and Environments", code TNMT.2023.04.01.*

REFERENCES

[1]. Z. Shen, S. Chen, *"A survey of automatic software vulnerability detection, program repair, and defect prediction techniques"*, Security and Communication Networks 2020 (1), 8858010 (2020).

[2]. L. Li, H. Feng, W. Zhuang, N. Meng, B. Ryder, *"Cclearner: A deep learning-based clone detection approach"*, pp. 249-260, (2017).

[3]. H. Wei, M. Li, *"Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code"*, pp. 3034-3040, (2017).

[4]. M. White, M. Tufano, C. Vendome, D. Poshyvanyk, *"Deep learning code fragments for code clone detection"*, pp. 87-98, (2016).

[5]. N. Marastoni, R. Giacobazzi, M. Dalla Preda, *"A deep learning approach to program similarity"*, pp. 26-35, (2018).

[6]. A. Sheneamer, *"CCDLC detection framework-combining clustering with deep learning classification for semantic clones"*, pp. 701-706, (2018).
[7]. A. Sheneamer, H. Hazazi, S. Roy, J. Kalita, *"Schemes for labeling semantic code clones using machine learning"*, pp. 981-985, (2017).
[8]. N. Shalev, N. Partush, *"Binary similarity detection using machine learning"*, pp. 42-47, (2018).
[9]. G. Zhao, J. Huang, *"Deepsim: deep learning code functional similarity"*, pp. 141-151, (2018).
[10]. X. Ban, S. Liu, C. Chen, C. Chua, *"A performance evaluation of deep-learnt features for software vulnerability detection"*, Concurrency and Computation: Practice and Experience 31(19), e5103, (2019).
[11]. A.G. Bacudio, X. Yuan, B.-T.B. Chu, M. Jones, *"An overview of penetration testing"*, International Journal of Network Security & Its Applications 3(6), 19, (2011).
[12]. A. Amos-Binks, J. Clark, K. Weston, M. Winters, K. Harfoush, *"Efficient attack plan recognition using automated planning"*, pp. 1001-1006, (2017).
[13]. W. Wang, D. Sun, F. Jiang, X. Chen, C. Zhu, *"Research and challenges of reinforcement learning in cyber defense decision-making for intranet security"*, Algorithms 15(4), 134, (2022).
[14]. M. Bhme, C. Cadar, A. Roychoudhury, *"Fuzzing: Challenges and reflections"*, IEEE Software 38(3), 79-86, (2020).
[15]. J. Li, B. Zhao, C. Zhang, *"Fuzzing: a survey"*, Cybersecurity 1, 1-13, (2018).
[16]. M. Alqaradaghi, T. Kozsik, *"Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code"*, IEEE Access (2024).
[17]. K. Abdulghaffar, N. Elmrabit, M. Yousefi, *"Enhancing Web Application Security through Automated Penetration Testing with Multiple Vulnerability Scanners"*, Computers 12(11), 235, (2023).
[18]. S. Alazmi, D.C. De Leon, *"A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners"*, IEEE Access 10, 33200-33219, (2022).
[19]. White Box Testing for Web Applications, 2020. https://www.offsec.com/blog/white-box-testing-web-applications/. (2024).
[20]. A. O'Mara, I. Alsmadi, A. Aleroud, D. Alharthi, *"Phishing Detection Based on Webpage Content: Static and Dynamic Analysis"*, pp. 39-45, (2023).
[21]. A. Aggarwal, P. Jalote, *"Integrating Static and Dynamic Analysis for Detecting Vulnerabilities"*, pp. 343-350, (2006).

TÓM TẮT

Tổng quan về phân tích tĩnh và phân tích động trong kiểm tra bảo mật ứng dụng

Trong bối cảnh các hệ thống thông tin ngày càng trở nên phức tạp và đối mặt với nhiều mối đe dọa an ninh mạng, việc đánh giá an toàn thông tin (ATTT) trở nên vô cùng quan trọng. Bài báo này tập trung vào hai phương pháp đánh giá ATTT phổ biến là đánh giá tĩnh và đánh giá động. Đánh giá tĩnh phân tích mã nguồn hoặc mã nhị phân để phát hiện lỗ hổng bảo mật từ giai đoạn phát triển phần mềm. Đánh giá động kiểm tra bảo mật của hệ thống trong quá trình hoạt động, giúp phát hiện lỗ hổng trong thời gian chạy. Bài báo trình bày tổng quan các kỹ thuật và công cụ của cả hai phương pháp, đồng thời so sánh ưu nhược điểm của chúng. Đánh giá tĩnh giúp phát hiện lỗi sớm nhưng có thể bỏ sót các lỗi thời gian chạy, trong khi đánh giá động kiểm tra thực tế nhưng có thể gây gián đoạn hệ thống. Sự kết hợp của cả hai phương pháp mang lại hiệu quả tốt nhất trong việc đảm bảo ATTT.

Từ khoá: An toàn thông tin; Phân tích tĩnh; Phân tích động; Lỗ hổng bảo mật; Kiểm thử phần mềm.