Journal of Computing and Electronic Information Management
ISSN: 2413-1660 | Vol. 14, No. 3, 2024

Research on Software Vulnerability Detection Methods Based on Deep Learning

Jiayuan Zhu*

Qingdao No.2 Middle School of Shandong Province, Qingdao, China
* **Corresponding author Email:** 13906395663@163.com

Abstract: This paper aims to investigate software vulnerability detection methods based on deep learning to address the ever-growing challenges in software security. With the rapid development of information technology, software vulnerabilities have become the primary targets of cyberattacks, posing severe threats to economic, military, and social security. By analyzing the limitations of existing software vulnerability detection methods, this paper explores the potential applications of deep learning technology in this field. Through a literature review, relevant theories of software vulnerabilities are introduced, including the concept, types, and impacts of vulnerabilities. Subsequently, detailed descriptions of deep learning-based vulnerability detection methods are provided, encompassing Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and their variant LSTM, as well as the application of program slicing techniques in vulnerability detection. By evaluating and improving existing deep learning models, a novel deep learning-based vulnerability detection framework is proposed, with its workflow and technical principles elaborated. The proposed deep learning-based vulnerability detection method can effectively enhance the accuracy and efficiency of software vulnerability detection, reduce reliance on expert knowledge, and lower human resource costs. Experimental results demonstrate that this method performs exceptionally well on multiple datasets, indicating broad application prospects and significant research value. The deep learning-based software vulnerability detection method offers new insights and tools for addressing current software security challenges. In the future, with the continuous development and improvement of deep learning technology, this method will play an even more crucial role in the field of software vulnerability detection.

Keywords: Deep learning; Software vulnerability; Detection technology.

1. Introduction

With the rapid development of information technology, software plays a vital role in various fields such as economics, military, and society. However, the potential security issues in software have also become increasingly prominent, posing a new global challenge. Software vulnerabilities are a primary cause of these security issues. Hackers can exploit these vulnerabilities to carry out various malicious attacks, such as stealing users' personal information or disabling critical infrastructure [1]. In modern society, users worldwide frequently interact and communicate, and if hackers choose a moment to launch attacks on operating systems or applications, the consequences would be immeasurable. According to statistics released by the Common Vulnerabilities and Exposures (CVE) organization, the number of software vulnerabilities discovered in 2000 was less than 4,600, but this figure surged to over 15,000 after 2017, and now exceeds 20,000 software security vulnerabilities, posing an even greater threat to computer users who utilize network services.

In the past few years, the frequency of cyberattacks on industries and businesses has continued to increase, intensifying the security threats faced by enterprises and individuals. The potential security issues in software not only plague developers but also directly relate to national cybersecurity. Recent high-profile vulnerabilities have demonstrated their potentially catastrophic consequences, affecting multiple economic and social aspects. For instance, the Wanna Decryptor emerged in 2017 [2], a ransomware that propagated through a "worm-like" mechanism, exploiting the "EternalBlue" vulnerability leaked by the NSA (National Security Agency of the United States). It became one of the most destructive and widespread vulnerabilities in recent years. This virus could achieve remote code execution through its designed network packets, encrypting critical files on users' computers and demanding Bitcoin payments for decryption. Despite the efforts of academia and industry to improve software quality, due to various factors such as developers or development environments, software vulnerabilities are almost impossible to completely avoid, and resolving them once they occur can be extremely challenging. Given the above, rapidly detecting vulnerabilities in software is of paramount importance. Currently, many detection methods still rely on manual review, which not only requires expert knowledge and extensive experience but also imposes significant pressure on human reviewers. Any oversight or omission could lead to undetected or misreported vulnerabilities, which, if ignored or misjudged, are highly likely to be exploited maliciously, resulting in severe consequences.

2. Related Theories of Software Vulnerabilities

2.1. Concept of Software Vulnerabilities

Software vulnerabilities refer to security flaws, faults, or weaknesses in software code that can be exploited by attackers [3]. These vulnerabilities can be maliciously utilized by unauthorized users. In brief, software vulnerabilities can be broadly classified into three major categories: software errors, software defects, and software failures. Software errors often stem from human factors, such as oversights or mistakes made by programmers during code writing,

resulting in software failing to meet its intended design objectives. The formation of software defects, on the other hand, is more widespread and encompasses not only human errors but also other non-human, objective conditions. These defects manifest during software operation and may introduce issues beyond the original design. Software failures, meanwhile, refer to instances where software fails to execute its predetermined functions, thereby unable to meet user or system requirements. Whenever new vulnerabilities are identified, relevant organizations classify them based on their nature, assign a unique identifier, and then record this information in specialized vulnerability databases for public reference by the security community. Lists of vulnerabilities in these databases, such as the Common Weakness Enumeration (CWE) and Common Vulnerabilities and Exposures (CVE), are widely used resources in the industry to identify and track known software security vulnerabilities.

2.2. Types of Software Vulnerabilities

common types of software vulnerabilities include: Buffer Overflow (CWE-787), Cross-Site Scripting (CWE-79), SQL Injection (CWE-89), Improper Input Validation (CWE-20), Out-of-Bounds Read (CWE-125), Use After Free (CWE-416), Cross-Site Request Forgery (CSRF, CWE-352), Integer Overflow or Wraparound (CWE-190), among others.

Buffer Overflow (CWE-787): Occurs when data is written outside the intended buffer boundaries, potentially leading to data corruption or code execution. Software can modify indices or perform pointer arithmetic to reference memory locations beyond buffer boundaries, and subsequent write operations may produce undefined or unexpected results.

Cross-Site Scripting (CWE-79): Arises when software fails to properly restrict user input and embeds it onto web pages served to other users. When malicious scripts are injected, attackers can perform various malicious operations, such as stealing victims' private information or forging their requests.

SQL Injection (CWE-89): Happens when software fails to properly filter user input, allowing attackers' inputs to be mistakenly interpreted as SQL statements. These can be used to alter query logic, bypass security checks, modify backend databases, or execute system commands.

Improper Input Validation (CWE-20): Occurs when software does not correctly validate received inputs or data for attributes necessary for safe and correct processing. Input validation is a common security practice to ensure that inputs are safe when being processed or communicated with other components.

Out-of-Bounds Read (CWE-125): Involves reading data outside the intended buffer boundaries, which could enable attackers to retrieve sensitive information from other memory locations or cause crashes. When code reads a variable amount of data, a lack of sentinel conditions to stop reading beyond buffer boundaries can lead to buffer overflow errors. Software can modify indices or perform pointer arithmetic to reference memory locations beyond buffer boundaries, and subsequent read operations may produce undefined or unexpected results.

Use After Free (CWE-416): Refers to referencing memory after it has been freed, which can lead to system crashes, the use of unexpected values, or the execution of arbitrary code.

Cross-Site Request Forgery (CSRF, CWE-352): Involves attackers forging user requests, exploiting a lack of proper mechanisms in web servers to verify that requests originate from users' genuine intentions.

Integer Overflow or Wraparound (CWE-190): Typically occurs when an integer value grows too large to be stored properly, causing it to wrap around to a very small number or a negative number. When user input triggers integer overflows, it can lead to security issues. In the subsequent experimental sections of this paper, publicly available datasets based on CWE classifications will be used to experimentally validate the models.

3. Vulnerability Detection Methods Based on Deep Learning

In recent years, deep learning techniques have found widespread applications in fields such as image processing and natural language processing (NLP), spurring the exploration of whether these techniques can be leveraged for software vulnerability detection. Through deep learning, potential features within code can be learned to identify the presence of security vulnerabilities in code [4]. Deep learning methods require researchers to provide generalized characteristics of software vulnerabilities, while more nuanced features are autonomously discovered by the models. If successfully applied to software vulnerability detection, deep learning could reduce reliance on traditional rule-based detection methods that heavily rely on experts, significantly lowering human resource costs. Recent research indicates that utilizing deep learning for software vulnerability detection is feasible; however, due to the unique structural characteristics of software code, compact and ordered code segments do not always exhibit distinct features readily captured by deep learning models. Consequently, the application of vulnerability pattern-based methods poses more complex challenges than problems in other deep learning domains, such as object detection [5], which involves identifying specific objects in an image based on known information and providing a confidence score for the identified object [6].

With the advancement of artificial intelligence, deep learning, as a crucial branch of machine learning, has been widely adopted in NLP and computer vision, among other fields, igniting researchers' interest in applying it to security vulnerability detection. According to Google Scholar statistics, 92 papers on vulnerability detection using deep learning were published in 2019 alone.

In traditional machine learning approaches, the extracted features often fail to comprehend semantic differences across datasets, meaning that the performance of detection models built using traditional methods can significantly degrade when confronted with different datasets. To address this issue, Song Wang [7] et al. developed a deep belief network (DBN) model that automatically learns semantic features from source code changes and label vectors extracted from programs. They utilized abstract syntax trees (ASTs) to establish file-level vulnerability detection models and leveraged source code changes to construct change-level detection models.

Zhen Li et al. proposed VulDeePecker, a vulnerability detection method based on a bidirectional long short-term memory (BLSTM) model [8]. VulDeePecker employs "code gadgets," which are sequences of statements describing variable flows and data dependencies, consisting of semantically related lines of code from the source code. In VulDeePecker, code's semantic relationships are classified as data dependencies or control dependencies. The authors used a commercial tool, Checkmarx, to extract library and API function calls related to data or control flows and their

corresponding program snippets. These snippets were then converted into token sequences, transformed into fixed-length vector representations using Word2vec technology, and finally fed into a BiLSTM model for vulnerability detection. VulDeePecker operates at the snippet level. To evaluate its performance, the authors conducted a series of experiments on open-source projects and the SARD dataset, revealing that VulDeePecker outperforms other static analysis tools in vulnerability detection. However, it has limitations: it can only process C/C++ programs and is limited to detecting vulnerabilities related to library/API function calls. Furthermore, the evaluation datasets were relatively small, encompassing only two types of vulnerabilities.

Subsequently, Zhen Li et al. furthered their research based on VulDeePecker by developing SySeVR, a deep learning-based vulnerability detection framework [9]. SySeVR aims to find a better way to represent programs as vectors. It extracts syntax vulnerability candidates (SyVCs) from the target program using ASTs and constructs data flow graphs (DFGs) and control flow graphs (CFGs) through program slicing techniques to build a program dependence graph (PDG) for the target code. The PDG is then used to transform SyVCs into semantic vulnerability candidates (SeVCs), preserving the syntactic and semantic information of the source code. Word2Vec is employed to generate embedding vectors, which are then used to train detection models through various RNNs (BLSTM and BGRU). SySeVR operates at the SeVC level, focusing on multi-line code segments. Experimental results show that SySeVR outperforms some state-of-the-art vulnerability detection methods on open-source projects and the SARD dataset, but it is also limited to detecting vulnerabilities in C/C++ code.

To overcome the issue of insufficient labeled data in datasets, Lin et al. [10] proposed a detection framework utilizing two independent BLSTM networks to extract features from two distinct types of data sources. This framework allows each network to train independently as a feature extractor. The two BLSTM networks learn different vulnerability patterns from two vulnerability datasets (SARD and real-world datasets). To bridge the gap between real-world samples and synthetic samples from the SARD project, the authors adopt different feature representation methods for different types of vulnerability samples: AST extraction for real-world samples to construct features and direct source code usage for synthetic samples. During training, one network inputs ASTs, while the other inputs source code. They argue that compared to networks trained on a single dataset, the dual-network approach can capture more "vulnerability knowledge" from both vulnerability datasets. Experimental results also demonstrate that models trained on two vulnerability datasets are more effective than those trained on only one.

4. Vulnerability Detection Techniques Based on Deep Learning

Deep learning is a crucial branch of machine learning. With the continuous advancement of artificial intelligence in recent years, deep learning technologies have also been continuously updated and developed, extending their applications to various fields. Currently, deep learning has been widely used in tasks such as computer vision, data mining, and natural language processing. Naturally, deep learning also finds ample applications in the field of software vulnerability detection [11].

4.1. Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN), first introduced by Yann LeCun et al. in the late 1980s and early 1990s, play a vital role in the deep learning domain due to their ability to process high-dimensional data, especially image and video data [12]. It was the first deep learning model successfully applied to computer vision tasks. Initially, CNN's applications were limited due to slow computer processing speeds, which hindered the training of large-scale deep neural networks. However, with the development of Graphics Processing Units (GPUs) and advancements in deep learning algorithms, CNNs have now been widely adopted in crucial domains like computer vision, natural language processing, and data mining. Additionally, CNNs comprise convolutional layers for feature extraction, pooling layers for simplifying feature maps, fully connected layers for classification and regression, and activation functions. Compared to other shallow neural networks, CNNs require fewer parameters. Typically, a CNN architecture encompasses Convolutions, Subsampling (Pooling), Full Connection layers, and activation functions.

4.2. Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) [13] are among the most significant types of neural networks. By introducing factors that influence sequential relationships, such as time order, RNNs can be viewed as multi-layer perceptrons unfolded over time, with an identical neural network structure at each time step. This characteristic enables RNNs to process time-series data with certain advantages. Due to their structural properties, RNNs are widely used in natural language processing and signal processing. Next, we will introduce Long Short-Term Memory Networks (LSTM), a special type of RNN extensively employed in software vulnerability detection. First proposed by Hochreiter & Schmidhuber in 1997, LSTM addresses the issues of short-term memory and gradient explosion that RNNs often encounter. Similar to RNNs, LSTM utilizes three types of gates—forget gate, input gate, and output gate—to control the cell state, or information flow.

4.3. Program Slicing

Program slicing is a program analysis technique that aims to assist programmers in efficiently locating and resolving issues by isolating specific parts of the program. This technique was first proposed by Mark Weiser [14] in 1979, with the objective of aiding programmers in understanding and diagnosing errors in programs.

The emergence of program slicing primarily responds to the challenges posed by increasing program complexity. As software scales up and program complexity grows, debugging becomes more intricate. Program slicing simplifies these problems by extracting specific parts of the program, enabling programmers to comprehend program logic more clearly and swiftly resolve issues. Moreover, program slicing plays a crucial role in software security. It helps developers quickly pinpoint security vulnerabilities in programs, such as buffer overflows or SQL injections. By slicing the program, developers can analyze the sliced code to rapidly identify vulnerable sections and take measures for remediation. At the same time, program slicing techniques are also valuable for security researchers and attackers in analyzing vulnerabilities and attack surfaces within programs. Through slicing, they

can quickly locate vulnerabilities and conduct in-depth analyses to better understand their nature and potential attack vectors.

Currently, program slicing techniques are primarily implemented through two approaches: static slicing and dynamic slicing. Dynamic slicing generates program slices by running the program, leveraging runtime information like variable values and function call stacks to determine program execution paths and variable values. During program execution, dynamic analysis methods can record the execution trace and runtime state, thereby enabling program slicing.

In contrast, static slicing generates program slices by analyzing and understanding program code. The control flow and data flow information in programs are crucial for static slicing. Control flow information refers to the execution order between statements in a program, while data flow information involves the data transfer relationships between variables in the program. Based on control flow and data flow information, static analysis methods can transform the program slicing problem into a path problem or a data dependency problem. The path problem focuses on finding paths from program inputs to outputs, while the data dependency problem concerns identifying input variables and statements related to output results.

5. Conclusion

As the complexity and scale of software systems continue to grow, the variety and quantity of software vulnerabilities are also on the rise, posing significant challenges to information security. This paper delves into the pressing challenges and the importance of current software vulnerability detection. With the sharp increase in the number of software vulnerabilities, traditional manual review methods have become increasingly unable to meet the demands for efficient and accurate detection. The introduction of deep learning technology has provided innovative solutions to this problem. Deep learning models, such as Deep Belief Networks (DBNs) and Bidirectional Long Short-Term Memory networks (BLSTMs), can automatically learn and extract complex features from software code, effectively identifying potential vulnerabilities. These methods not only reduce the reliance on expert knowledge but also significantly improve the efficiency and accuracy of vulnerability detection. However, current research still has some limitations, primarily focusing on vulnerability detection in C/C++ languages and with relatively small datasets available. Looking ahead, with the continuous development and improvement of deep learning technology, it is anticipated that software vulnerability detection methods based on deep learning will become more mature and widespread. By continuously optimizing model structures and algorithms, expanding dataset sizes, and exploring multi-language, multi-type vulnerability detection strategies, we can expect to build a more efficient and intelligent vulnerability detection system, thereby further enhancing software security and safeguarding the rights and data security of a broad range of users.

References

[1] Shen Z, Chen S. A Survey of Automatic Software Vulnerability Detection, Program Repair, and Defect Prediction Techniques[J]. Security and Communication Networks, 2020, 2020.

[2] Bistarelli S, Parroccini M, Santini F. Visualizing Bitcoin Flows of Ransomware: WannaCry One Week Later[C]//ITASEC. 2018.

[3] Dempsey K, Takamura E, Eavy P, et al. Automation Support for Security Control Assessments: Software Vulnerability Management[R]. National Institute of Standards and Technology, 2020: 93.

[4] Chakraborty S, Krishna R, Ding Y, et al. Deep learning based vulnerability detection: Are we there yet[J]. IEEE Transactions on Software Engineering, 2021.

[5] Papageorgiou C P, Oren M, Poggio T. A general framework for object detection[C]//Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271). IEEE, 1998: 555-562.

[6] Zhang Weiguo. A Method for Software Vulnerability Detection Based on Code Semantic Vector Representation and Deep Learning [D]. Harbin Institute of Technology, 2020.

[7] Wang S, Liu T, Nam J, et al. Deep semantic feature learning for software defect prediction[J]. IEEE Transactions on Software Engineering, 2018, 46(12): 1267-1293.

[8] Li Z, Zou D, Xu S, et al. Vuldeepecker: A deep learning-based system for vulnerability detection[J]. arXiv preprint arXiv:1801.01681, 2018.

[9] Li Z, Zou D, Xu S, et al. Sysevr: A framework for using deep learning to detect software vulnerabilities[J]. IEEE Transactions on Dependable and Secure Computing, 2021, 19(4): 2244-2258.

[10] Lin G, Zhang J, Luo W, et al. Software vulnerability discovery via learning multi-domain knowledge bases[J]. IEEE Transactions on Dependable and Secure Computing, 2019, 18(5): 2469-2485.

[11] Mao Y, Li Y, Sun J, et al. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks[C]//2020 IEEE International Conference on Big Data (Big Data). IEEE, 2020: 4651-4656.

[12] Goodfellow I, Bengio Y, Courville A, et al. Deep Learning, vol. 1. Cam bridge: MTT press[J]. 2016.

[13] Schmidhuber J. Deep learning in neural networks:An overview[J].Neural networks,2015, 61: 85-117.

[14] Weiser M D. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method[M]. University of Michigan, 1979.