國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

使用深度學習技術進行自動化的程式碼弱點偵測

Automated Vulnerable Code Detection Using Deep Learning Technique

柯以恆

Yi-Heng Ko

指導教授: 王凡 博士

Advisor: Farn Wang, Ph.D.

中華民國 113 年 8 月

August 2024

doi:10.6342/NTU202401783

誌謝

首先，我要衷心感謝我的指導教授王凡教授。在整個論文寫作過程中，給予我寶貴的指導和建議，無論在研究方向的選擇上，還是在論文的撰寫和修改上，都給予了我極大的幫助，使我在學術上獲益良多。同時，我也要感謝實驗室的同學們。謝謝你們在我遇到困難時給予的支持和幫助，感謝你們在科研過程中與我分享的經驗和知識。特別感謝實驗室王冠彥同學。最後，我要感謝我的家人和朋友。謝謝你們在我攻讀學位期間給予的無私支持和鼓勵，使我能夠專心投入到學術研究中。

doi:10.6342/NTU202401783

摘要

為了確保應用程式中不存在能夠被有心人士利用的漏洞，程式碼安全檢測在軟體開發中一直扮演一個重要角色。傳統的程式碼安全測試通常依賴手動檢查或基於規則的方法，這樣的方法可能相當耗時且容易出現人為錯誤。近年來隨著自然語言處理的發展，深度學習儼然成為程式碼安全測試的一種手段，我們將在這篇論文研究將深度學習技術應用於程式碼安全測試的可能性，目標是能夠提高軟體開發流程中安全分析的效率和效力。在本篇研究中，我們以長短期記憶模型作為模型架構對資料集進行訓練，並測試了兩種嵌入方法在生成程式碼向量表示上的效能以提高訓練效率。此外，我們還將在 GitHub 上蒐集的多個專案應用於論文中所提出的模型上，再把掃描結果與現有的靜態測試工具做比較並對其性能進行評估，結果顯示我們的研究成果比起市售的靜態安全測試軟體能達到更好的表現，最後透過分析實驗的數據，提出可能改進的方法。

關鍵字：

靜態分析，弱點偵測，程式碼嵌入，深度學習，自然語言處理

doi:10.6342/NTU202401783

ABSTRACT

To avoid the existence of exploitable vulnerabilities within applications, security testing has always played a crucial role in software development. Traditional code security testing methods often rely on manual inspection or rule-based approaches, which can be time-consuming and prone to human error. With the recent advancements in natural language processing, deep learning has emerged as a viable approach for code security testing. In this thesis, we investigate the application of deep learning techniques to code security testing with the aim of enhancing the efficiency and effectiveness of security analysis in the software development process. In our study, we train our dataset using a Long Short-Term Memory (LSTM) model as the architecture and evaluate the performance of two embedding methods in generating code vector representations to increase training efficiency. Additionally, we apply our proposed models to multiple projects collected from GitHub, compare the scan results with existing static testing tools, and evaluate their performance. The results demonstrate that our research outcomes are perform better than commercially available static application security testing (SAST) tools. Through the analysis of experimental data, we propose potential improvements and future work for research.

Keywords:

Static analysis, Vulnerability detection, Code embedding, Deep learning, Natural language processing

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

CONTENTS

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

LIST OF FIGURES

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

LIST OF TABLES

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

Chapter 1

Introduction

1.1 Background

The ever-growing complexity of software system and increasing prevalence of cyber-attack have raised the importance of software security testing. Vulnerabilities in software code can expose applications to a wide range of security threats, including data breaches, system compromises, and unauthorized access. The well-known Shellshock bash vulnerability was costed by the code flaw in Bash, affecting tons of system and server that worked on Unix-based operating system [1]. According to Common Vulnerabilities and Exposures database [2], thousands of such vulnerabilities are reported publicly every year. The report also indicated that there were 2581 vulnerabilities in 2023 of code execution. Compared with 2022, it is an increase of 24.8%. These vulnerabilities are often caused by subtle errors made by programmers and can propagate quickly due to the prevalence of open-source software and code reuse [3], therefore, it is a significant issue that we cannot ignore and have to deal with.

There are two major strategies of traditional software vulnerability detection (SVD), which are static and dynamic analysis. Static analysis involves examining the source code, bytecode, or binary code of an application without executing it. This method aims to identify vulnerabilities, coding errors, and other potential security issues early in the development process. However, static analysis has its limitations. It may produce a high number of false positives [4][5], where non-existent issues are reported, leading to wasted effort in investigating them. Moreover, static analysis might miss certain runtime-specific vulner-

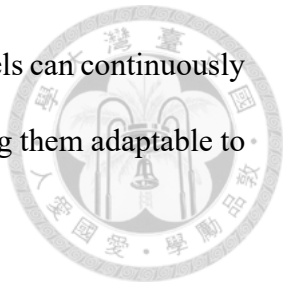
doi:10.6342/NTU202401783

abilities, such as those related to the environment or dynamic dependencies [6], and also rely on domain experts to pre-define the error features, imposing intense manual labor.

Dynamic analysis on the other hand, involves testing an application while it is running. This approach simulates real-world attacks by providing various inputs and monitoring the application's behavior to identify vulnerabilities. A key advantage of dynamic analysis is its ability to test the application in a realistic environment, which helps in identifying vulnerabilities that static analysis might miss.Additionally, dynamic analysis tools are often easier to set up and use compared to static analysis tools. Despite its benefits, dynamic analysis has several disadvantages. One major drawback is limited code coverage, as it relies on specific test cases and conditions to trigger vulnerabilities. This means that some code paths may remain untested, especially in complex applications. Dynamic analysis can also be resource-intensive, requiring significant computational power and time to perform thorough testing. Additionally, reproducing and diagnosing issues found during dynamic analysis can be challenging, as it may not always be clear which part of the code is responsible for the observed behavior [7].

Recent research has demonstrated the potential of ML in improving the accuracy and efficiency of vulnerability detection. For instance, machine learning models such as neural networks, decision trees, and support vector machines have been trained on large datasets of vulnerable and non-vulnerable code to automatically identify security issues [8]. These models can analyze code semantics and patterns that traditional static and dynamic analysis might miss, providing a more nuanced understanding of potential vulnerabilities. Furthermore, machine learning approaches can be particularly effective in handling the code's variability and complexity. Techniques such as unsupervised learning and anomaly detection can identify unusual patterns or behaviors in the code that may in-

doi:10.6342/NTU202401783



dicate previously unknown vulnerabilities [4]. Additionally, ML models can continuously improve their detection capabilities by learning from new data, making them adaptable to the evolving landscape of software vulnerabilities.

However, the application of machine learning in vulnerability detection is not without challenges. One major issue is the need for large, high-quality datasets to train effective models. The scarcity of labeled vulnerability data can hinder the performance of ML models, leading to issues such as overfitting and poor generalization to unseen code [9]. Additionally, machine learning models can also generate false positives and false negatives, similar to traditional analysis methods, and require careful tuning and validation to ensure reliability.

1.2 Motivation

Recent research has demonstrated the significant potential of machine learning (ML) in enhancing vulnerability detection[10][11][12][3][13]. ML models, such as neural networks, decision trees, and support vector machines, have shown promising results by being trained on large datasets of vulnerable and non-vulnerable code, thereby automating the identification of security issues. These models can analyze code semantics and patterns that traditional static and dynamic analysis methods might overlook, providing a more nuanced understanding of potential vulnerabilities .

However, the application of ML in vulnerability detection is not without its challenges. One major issue is the quality of the dataset used for training ML models. High-quality, labeled datasets are crucial for the effective performance of ML models. The scarcity of such datasets, especially those accurately labeled for vulnerability detection, can hinder the performance of ML models, leading to problems such as overfitting and

doi:10.6342/NTU202401783

poor generalization to unseen code .

Another challenge lies in the lack of focus on evaluating the impact of hyperparameter tuning in previous research. Hyperparameters significantly influence the performance of ML models. While some studies have achieved commendable results, there is a notable gap in comprehensive investigations into how different hyperparameter configurations can optimize model performance for vulnerability detection .

Additionally, the majority of existing research has not adequately addressed the needs of the Python programming language, despite its widespread popularity in the software development community. Python's extensive use in various applications and its unique syntax and semantics necessitate targeted research to develop effective vulnerability detection models tailored specifically for Python code . Current studies have predominantly focused on other programming languages, leaving a critical gap in the understanding and addressing of vulnerabilities within Python applications.

Therefore, this thesis aims to address these challenges by focusing on the quality and labeling of datasets, exploring the impact of hyperparameter tuning, and specifically targeting Python for vulnerability detection. By doing so, we hope to contribute to the development of more accurate and efficient ML models for identifying vulnerabilities in Python code, ultimately enhancing the overall security of software systems.

1.3 Contribution

In this thesis, we present a deep learning-based vulnerability detection model that can automatically learn features of vulnerable code from a large, real-world codebase. For model training, we leverage the Simulated Annealing (SA) algorithm to explore the ideal

doi:10.6342/NTU202401783

hyperparameters. To ensure that CodeBERT is suitable for the source code embedding task, we also implement the traditional word embedding method Word2Vec [14] as our baseline. At the end of our study, we apply our model to real-world projects. To justify our achievements, we also implement the open-source static application security testing (SAST) tool Bandit, and the commercial tool Checkmarx for comparison. In summary, there are four main contributions of this research:

1. We propose a deep learning model that can detect certain vulnerabilities in natural codebase
2. We leverage the Simulated annealing algorithm to explore the ideal hyperparameters for model
3. Comparing the performance of Word2Vec and CodeBERT embedding methods for source code embedding task
4. Evaluate the model performance by testing the real word projects that exist vulnerable issues, then comparing the result with existing SAST tools

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

Chapter 2

Related Work

In this section, we will discuss some academic research on vulnerability detection and industrial tools used for the same purpose. Based on previous studies, we focus on exploring any potential ways to yield better results.

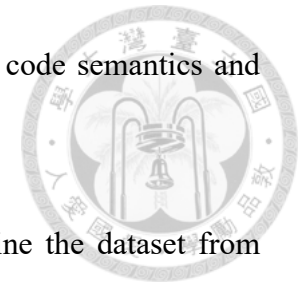
2.1 Machine Learning for Vulnerability Detection

Russell et al. [3] collect their dataset from Debian, Github and synthetic examples from the SATE IV Juliet test suite. All samples are written in C, labeling with three open source static analysis tools. They leverage CNN and RNN as feature-extraction approaches follow by a machine learning algorithm random forest to learn the source feature and classify the vulnerabilities, they yield a decent result on SATE IV by reaching 0.84 f1-score. However, the f1-score down to 0.566 when their model applied on the Debian and Github dataset, illustrated that they can work on synthetic dataset well but real-world dataset. On top of that, their approach can highlight the vulnerable part instead of mark the whole function as vulnerable was pretty impressive.

2.2 Deep Learning for Vulnerability Detection

Chakraborty et al. [12] systematically study different aspects of deep learning-based vulnerability detection to effectively find real-world vulnerabilities. By applying previous works to real-world cases, they found that the results are not as good as claimed. They further indicate some limitations and shortcomings of existing models and datasets, which include that the datasets are too basic, unable to represent the complexity of real-world

doi:10.6342/NTU202401783



scenarios, and the modeling techniques do not completely address code semantics and data imbalance in vulnerability detection.

Wartschinski et al. [13] follow the suggestion to directly mine the dataset from GitHub. They leverage Word2Vec as the embedding approach and use an LSTM model to learn the vulnerability features. Similar to our work, their model yielded good results in detecting vulnerable code. However, there are still some aspects to discuss. Firstly, regarding the embedding method, we consider that CodeBERT has the potential to perform better than Word2Vec. Secondly, the hyperparameters of the model are crucial to model training; their work only conducted a simple investigation on parameter tuning. We believe there is an opportunity to achieve better performance with deeper research on parameter tuning. Overall, this thesis provides substantial guidance for our work.

Recent research by Wang et al. [15] conducted a comprehensive study on different deep learning models for code vulnerability detection. Their work involved exploring various machine learning and deep learning models, alongside different embedding methods such as Word2Vec, FastText, and CodeBERT. The models evaluated included XGBoost, LSTM, GRU, MLP, and CNN. According to their findings, the combination of Word2Vec with LSTM demonstrated the best performance among all the tested configurations. However, upon examining their source code, it was discovered that the hyperparameters used were the same as those in previous studies[13]. This raises a critical discussion point: CodeBERT generates significantly more vector dimensions than Word2Vec for embeddings, which could potentially alter the results when different hyperparameter settings are applied.

doi:10.6342/NTU202401783

2.3 Static Application Security Testing Tool

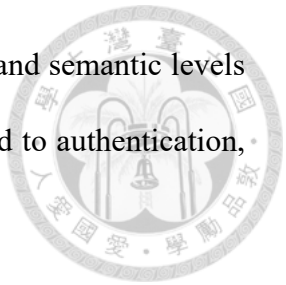
Static Application Security Testing (SAST) tools are designed to analyze source code, byte code, or binary code to identify potential security vulnerabilities. They are an essential component of a secure software development lifecycle.

Bandit [16] is an open source SAST tool, designed for Python to identify common issues and vulnerabilities, this tool now maintained by PyCQA (Python Code Quality Authority) Bandit first parses the Python source code provided to it. It utilizes the Python Abstract Syntax Tree (AST) module to parse the code into an abstract syntax tree representation. Once the source code is parsed, Bandit applies a set of predefined security rules to the abstract syntax tree. Each rule corresponds to a specific security issue or vulnerability pattern that Bandit is designed to detect. These rules are defined based on common security best practices and known vulnerabilities. Bandit analyzes the abstract syntax tree to identify patterns that match the criteria specified by the security rules. This involves examining the structure of the code, such as function calls, variable assignments, and control flow statements, to detect potential security.

Checkmarx [17] is a leading provider of application security testing solutions. Their primary offering is the Checkmarx Static Application Security Testing (SAST) solution, which is designed to help organizations identify and remediate security vulnerabilities in their software applications throughout the development lifecycle. Checkmarx performs static code analysis on the source code of an application, this analysis is typically performed during the development phase, but can also be integrated into continuous integration and continuous deployment (CI/CD) pipelines to automate security testing. The Checkmarx SAST solution scans the source code to identify security vulnerabilities, cod-

doi:10.6342/NTU202401783

ing errors, and compliance issues. It analyzes the code at the syntax and semantic levels to detect a wide range of security weaknesses, including those related to authentication, authorization, input validation, cryptographic usage, and more.



doi:10.6342/NTU202401783

Chapter 3

Preliminaries

3.1 Natural Language Processing

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human language. The main goal of NLP is to enable computers to understand, interpret, and generate human language. Over the years, NLP has evolved from simple rule-based systems to sophisticated models that leverage machine learning and deep learning techniques. These advancements have significantly improved the accuracy and capabilities of NLP applications, such as machine translation, sentiment analysis, and chatbots [18].

Recently, the introduction of large pre-trained language models like BERT and GPT has revolutionized NLP. These models are trained on vast amounts of text data and can capture complex patterns and relationships within the language [19][20]. As a result, they have achieved state-of-the-art performance on various NLP tasks.

3.2 Word Embedding

Word embeddings are a fundamental component in many Natural Language Processing (NLP) applications. They represent words in continuous vector spaces where semantically similar words are mapped to proximate points. This dense representation captures syntactic and semantic properties of words, facilitating better performance in various NLP tasks compared to traditional one-hot encoding methods [14].

The development of word embeddings began with models such as Word2Vec, which

doi:10.6342/NTU202401783

uses shallow neural networks to generate word vectors by predicting words in a given context or context words for a given word [21]. Another popular model, GloVe (Global Vectors for Word Representation), generates word embeddings by aggregating global word-word co-occurrence statistics from a corpus [22]. These advancements have significantly enhanced tasks like text classification, sentiment analysis, and machine translation by providing rich word representations [23].

3.3 Source Code Embedding

Source code contains substantial information that can be exploited by machine-learning algorithms. Prior approaches have utilized embedding methods like GloVe [22] and Word2Vec [14] to generate pre-trained vectors for singular tokens. Modern software development often prioritizes writing the human-readable source code. This leads to a situation where we can extract the semantic information from the source code using the techniques originally intended for the NLP [24][25]

CodeBERT [26] is pre-trained on a large corpus of bimodal data that includes both natural language (NL) and programming language (PL) text. This pre-training involves two key tasks: Masked Language Modeling (MLM): Similar to BERT, where random tokens are masked and the model learns to predict them, CodeBERT masks tokens in both code and natural language texts. This helps the model learn contextual relationships. Replaced Token Detection (RTD): This task involves replacing tokens with plausible alternatives and training the model to identify the correct token, further enhancing its understanding of code semantics. The training data for CodeBERT includes a diverse set of programming languages, including Python, allowing the model to understand various syntactic and semantic patterns in Python code.

doi:10.6342/NTU202401783

Chapter 4

Methodology

In this section, we present and discuss our approach. The overview of the methodology structure is illustrated in Fig. 4.1

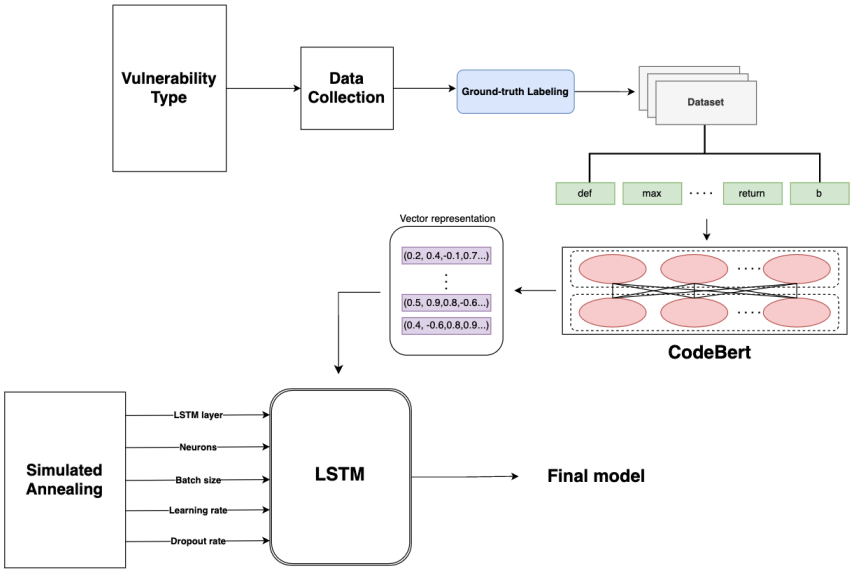


Figure 4.1: The overview of approach.

4.1 Vulnerability Type Selection

To choose the vulnerability types for model training, we refer to the following websites: OWASP Top 10 [27], Common Vulnerabilities and Exposures (CVE) Database [2], National Vulnerability Database (NVD) [28], and GitHub Security Advisories [29]. Our goal is to design a model that fits real-world scenarios; therefore, the vulnerabilities must be common. Furthermore, to ensure the model can learn the vulnerable patterns, the vulnerabilities must contain some semantic and syntactic features. Lastly, to cover a sufficient amount of training data, it is necessary to have a significant quantity of vulnerable commits on GitHub. Based on the points mention above, we summarized the considerations

doi:10.6342/NTU202401783

below:

1. Common vulnerability

2. Vulnerability has syntactic feature to learn

3. Enough commits from Github

4.2 Dataset Collection

We utilized GitHub as our data source due to its extensive collection of open-source repositories and the patch system, which enables us to access previous versions of source code potentially containing unfixed vulnerabilities.The initial step involves gathering a substantial number of commits that are potential vulnerability fixes. This was achieved by querying GitHub data using its public REST API to identify commits that addressed vulnerabilities. Due to GitHub's constraints, we first compiled a dataset of commits in different languages. Since our focus was solely on the Python language, samples not written in Python or written in multiple languages (e.g., JavaScript, HTML) were filtered out.

After collecting the commits written in Python, the next step was to filter out commits unrelated to vulnerabilities and code changes based on vulnerability-related keywords like "SQL injection," "XSS," "fixed," "prevent," etc.. The final step was to download all the commits and their details; we used the PyDriller tool to assist in this task. As mentioned above, we ended up gathering approximately 200k lines of code (LOC) and over 1.2k commits from more than 500+ different Python repositories.

doi:10.6342/NTU202401783

4.3 Ground-truth Labeling

After filtering the commits based on their messages and downloading the relevant code changes in the form of diffs, the collected data contain information from the previous version of the code, indicating that the vulnerable part had not been fixed. For example, commit messages like "SQL injection prevention" or "XSS vulnerability fixed" imply that the changed part could be vulnerable and has been addressed. We leverage this property to complete the labeling task. We created the final labeled dataset as follows. We first remove the comments from the code, because it is unlikely to cause a vulnerability. After that, we conduct a similar procedure to the work of Hovsepyan et al. [30] and Wartschinski et al. [13]. A small focus window traverses the source code in steps of length n, with surrounding context of roughly length m (where $m > n$). Overlapping blocks are generated, and if a block contains partially vulnerable code, it is labeled as vulnerable; otherwise, it is labeled as clean. Parameters n and m are optimized experimentally, with typical values being around 10 lines of code for capturing vulnerability context.

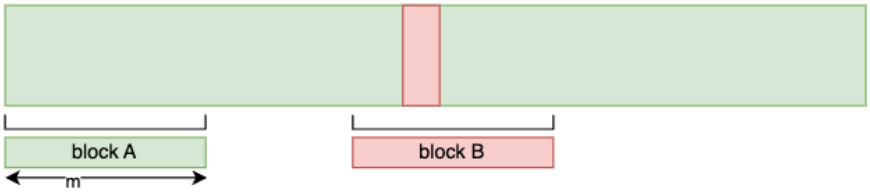


Figure 4.2: Splitting the whole code with vulnerable (red) and non-vulnerable (green) parts.

4.4 Embedding Layer

In this section, we briefly describe how CodeBERT generate source code embedding. The first thing CodeBERT do, is tokenizing the entire code snippet, for tokenizing the source code, CodeBERT uses Byte-Pair Encoding (BPE), a subword tokenization tech-

doi:10.6342/NTU202401783

nique. This method breaks down Python code into subwords rather than whole words or characters. For example, a variable name like add_one might be tokenized into ['add', '_', 'one']. BPE helps handle the variability and complexity of programming languages, reducing the number of out-of-vocabulary tokens. Once the Python code is tokenized, each token is converted into embeddings using the learned vocabulary and positional encodings. These embeddings are then fed into the transformer layers of CodeBERT. The model's self-attention mechanism allows it to capture dependencies and contextual relationships between different parts of the code. This is particularly useful for understanding Python' s indentation-based syntax and dynamic typing.

CodeBERT generates contextual embeddings by considering the entire input sequence, which means that each token's embedding is influenced by its surrounding tokens. This is crucial for understanding the semantics of Python code, where the meaning of a token can heavily depend on its context within functions, loops, or classes [26].

4.5 Hyperparameter Tuning of LSTM Model Using Simulated Annealing

Hyperparameter tuning is a critical step in optimizing the performance of machine learning models. For Long Short-Term Memory (LSTM) networks, the selection of hyperparameters such as the number of layers, number of units per layer(Neurons), learning rate, batch size, and dropout rate can significantly influence the model's ability to learn and generalize. In this section, we discuss the application of simulated annealing (SA) as a method for hyperparameter optimization in our LSTM model.

The number of neurons in an LSTM model affects its ability to learn from sequential

doi:10.6342/NTU202401783

data by determining its capacity to capture and retain information. More neurons generally enhance the model's ability to learn complex patterns and long-term dependencies, however, increasing the number of neurons also raises the computational cost and training time, and can lead to overfitting if the dataset is not large enough.

The batch size in training an LSTM model influences how many samples are processed before updating the model's weights. Smaller batch sizes generally allow the model to learn more detailed patterns since updates occur more frequently, however, too small a batch size can make training noisy and less stable. Larger batch sizes, on the other hand, provide more stable and accurate gradient estimates, which can lead to faster training times per epoch but may require more memory and could risk underfitting if the model updates too infrequently.

Dropout is a regularization technique used in LSTM models to prevent overfitting by randomly setting a fraction of input units to zero at each training step. This process helps to ensure that the model does not become too reliant on specific neurons and instead learns more robust features, however, using too much dropout can lead to underfitting, where the model fails to capture sufficient information from the training data.

4.5.1 Simulated Annealing Overview

Simulated annealing is a probabilistic optimization technique inspired by the annealing process in metallurgy. It aims to find a global minimum in a large search space by exploring potential solutions and gradually converging to an optimal set of parameters. The algorithm accepts both improvements and, with decreasing probability, worse solutions as it explores the parameter space. This allows the method to escape local minima

doi:10.6342/NTU202401783

and find a more optimal global solution over time.

The algorithm's name and function are derived from the process of heating and then slowly cooling a material to remove defects, thus optimizing its structure. Simulated annealing uses a temperature parameter that decreases over time, controlling the probability of accepting worse solutions as the search progresses.

4.5.2 Implementation of LSTM Hyperparameter Tuning

To implement simulated annealing for tuning our LSTM model's hyperparameters, we defined the following key components:

1. **Objective Function**: The objective function measures the model's performance, typically the validation loss or accuracy on a held-out dataset. For our experiments, we used the F1-score as the objective function.

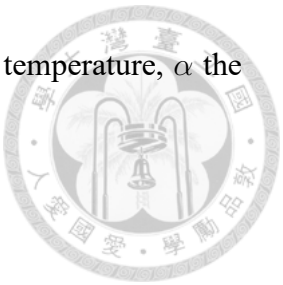
2. **Search Space**: The search space includes the ranges of hyperparameters to be optimized. For our LSTM model, we considered the following hyperparameters:

    - Neurons: [50, 100, 150, 200]
    - Learning rate: [0.0001, 0.001, 0.01, 0.1]
    - Batch size: [32, 64, 128, 256]
    - Dropout rate: [0.2, 0.4, 0.6]

3. **Initial Solution**: The initial set of hyperparameters was selected randomly within the defined search space.

4. **Temperature Schedule**: The temperature controls the acceptance probability of worse solutions. We used an exponential decay schedule where the temperature T

doi:10.6342/NTU202401783



decreases according to $T = T_0 \times \alpha^k$, with T_0 being the initial temperature, α the decay rate, and k the current iteration number.

5. **Neighborhood Function**: This function generates a new set of hyperparameters by slightly perturbing the current set. For instance, we could increase or decrease the number of units in an LSTM layer or adjust the learning rate within a small range.

6. **Acceptance Criterion**: The acceptance criterion is based on the Metropolis criterion. A new solution is always accepted if it has a better objective value. If it is worse, it is accepted with a probability P given by:

$$P = \exp\left(\frac{\Delta L}{T}\right) \tag{4.1}$$

where ΔL is the change in the objective function (validation loss) and T is the current temperature.

4.5.3 Conclusion

Simulated annealing proved to be an effective method for hyperparameter tuning of the LSTM model in our study. By allowing the exploration of suboptimal solutions and gradually focusing on the global optimum, this approach provided a robust mechanism for finding a well-performing set of hyperparameters. Future work could explore hybrid approaches that combine simulated annealing with other optimization techniques to further enhance the tuning process.

doi:10.6342/NTU202401783

4.6 LSTM Model

Deep-learning architecture such as Convolution Neural Network(CNN), Recurrent Neural Network(RNN), and Long-Short Term Memory(LSTM) [31] are suited to retrieve the semantic of code, while RNN and LSTM can capture much longer information than CNN due to the memory lost, research done by Wu, Fang, et al. [11] and Dam et al. [32], further indicated that LSTM perform better than CNN and RNN when it comes to the code feature extraction task, as a result, we apply LSTM as our model architecture. After pre-processing of the data, we can start training our LSTM model, we used Keras library [33] to construct the whole model architecture. LSTMs require sequences of fixed length as input. Since code can vary in length, sequences might be padded or truncated to ensure uniform length. As mention above, we applied sliding window approach to generate overlapping sequences from the code, ensuring that the model sees different parts of the code in context.

An LSTM-based model for code feature extraction typically includes the following layers: **LSTM Layers** to capture the sequential dependencies in the code, and **Dense Layers**, fully connected layers that map the learned features to the output space, depending on the task (e.g., classification, regression).

doi:10.6342/NTU202401783

Chapter 5

Experiment

In this section, we will conduct our experiment. After implementing the Simulated Annealing algorithm, we will use the results for LSTM model training. The next part will focus on evaluating two embedding methods: Word2Vec and CodeBERT. Drawing from the research conducted by Wartschinski et al. [13], who trained a Word2Vec model for Python source code embedding, we aim to utilize their findings as our baseline model and compare its performance with CodeBERT. In the final part, we will compare our model with the SAST tools Bandit and Checkmarx to analyze our model's performance.

We run our experiments on a server with a Nvidia RTX-3060 with 16 gigabytes of memory, and Intel i5-12400F CPU operating at 4.0 GHz.

5.1 Evaluation Metric

To evaluate the performance of our work, we employ four commonly used metrics: accuracy, precision, recall, and F1-score, which are derived from four key concepts: true positive (TP), false positive (FP), true negative (TN), and false negative (FN). In our study, TP and TN indicate correct predictions of "vulnerable" and "non-vulnerable" respectively, while FP and FN signify incorrect predictions of "vulnerable" as "non-vulnerable" and " non-vulnerable" as "vulnerable" respectively. Accuracy represents the overall correctness; however, in our case, accuracy alone may not offer significant insight due to dataset imbalance. Since the majority of our samples are clean and vulnerable parts are relatively

doi:10.6342/NTU202401783

rare, accuracy may be skewed towards a high value. The accuracy is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

The precision measures the proportion of positive predictions that are actually correct. The precision is defined as follows:

$$Precision = \frac{TP}{TP + TN} \tag{5.2}$$

The recall measures the proportion of actual positive data points that are correctly identified by the model. The recall is defined as follows:

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

The F1-score use a harmonic mean of precision and recall, providing a balanced view of both metrics. The F1-score is defined as follows:

$$F1 - Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5.4}$$

5.2 Training

5.2.1 Dataset

As per the considerations mentioned above, we chose four types of vulnerabilities: 'SQL Injection,' 'Cross-site Scripting,' 'Cross-site Request Forgery,' and 'Command Injection.' These samples were all mined from GitHub. We filtered out commits with either too many or too few lines of code (LOC), as such extremes are not conducive to effective model training. The dataset information is shown in the Table 5.1. We used an 80:20

doi:10.6342/NTU202401783

train-test split for model training.

	Repositories	Commits	Files	Code samples		Vulnerable Code Samples	
				Train	Test	Rate	Amount
SQL Injection	336	406	657	7413	1589	27.35%	460
XSS	39	69	81	2987	640	13.62%	88
Command Injection	85	106	197	4147	889	15.38%	134
XSRF	88	141	256	5228	1210	21.46%	237
Total	**548**	**722**	**1191**				

Table 5.1: Information of dataset.

5.2.2 Hyperparameters

Hyperparameters of our model are using the algorithm we mentioned above. The hyperparameters detail show in the Table 5.2.

Using simulated annealing, we optimized the hyperparameters of our LSTM model. The algorithm was run for 100 iterations, starting with an initial temperature $T_0 = 1.0$ and a decay rate $\alpha = 0.95$. The validation loss was evaluated at each iteration to monitor the performance of the model. Compared to grid search and random search methods, simulated annealing demonstrated more efficient exploration of the hyperparameter space, leading to better model performance within a shorter computational time. The final LSTM model, with the optimized hyperparameters, achieved a significant reduction in validation loss and improved generalization on unseen data.

	Value
Epochs	100
Neurons	200
Batch size	128
Dropouts	20%
Learning rate	1e-3
Optimizer	Adam

Table 5.2: Detail of hyperparameters.

doi:10.6342/NTU202401783

5.2.3 Result

In this section, we present our training results on four types of vulnerabilities. Table 5.3 depicts four metrics for each dataset. The LSTM model using CodeBERT as the embedding approach achieved, on average, an accuracy of 96.4%, a precision of 92.5%, a recall of 85.0%, and an F1-score of 88.6%. The SQL injection dataset yielded a relatively low F1-score compared to the other datasets. This is likely because the size of the SQL injection dataset is larger than the others, potentially containing more noisy or low-quality data, we will discuss this further later. Overall, we achieved a decent result on the vulnerability detection task.

Vulnerability	Accuracy	Precision	Recall	F1-score
SQL Injection	92.5%	86.2%	80.0%	83.1%
XSS	98.0%	95.2%	84.9%	89.7%
XSRF	98.1%	94.5%	86.3%	90.2%
Command Injection	97.3%	94.3%	89.1%	91.6%
Average	**96.4%**	**92.5%**	**85.0%**	**88.6%**

Table 5.3: Training results.

5.3 Evaluation

To evaluate the effectiveness of CodeBERT as an embedding method, we replicated the work of Wartschinski et al. [13], who utilized Word2Vec as an embedding method followed by an LSTM model to extract code features for Python source code vulnerability detection, which aligns with our objective. Since we do not have a pre-trained Python embedding method, we based on the dataset they provided and manually trained a Word2Vec model for the embedding task. The parameters of the Word2Vec model are set as follows:

- Training iterations: 100

doi:10.6342/NTU202401783

- Vector dimensionality: 300
- Minimum count: 10

The minimum count refers to the frequency of a token appearing in the training corpus, where tokens appearing less than 10 times will be disregarded. This allows us to eliminate rare variable names or strings that are irrelevant. After training Word2Vec, we have successfully reproduced the same testing results as their dataset. This means we can now apply our model to the same dataset to evaluate its performance.

Fig. 5.1 illustrates the results of Word2Vec and CodeBERT embeddings applied to a large Python corpus. We utilized t-SNE (t-distributed Stochastic Neighbor Embedding), a dimensionality reduction technique commonly used for visualizing high-dimensional data. This technique is particularly beneficial for visualizing datasets with numerous features (dimensions) in a two-dimensional (2D) or three-dimensional (3D) space, facilitating human interpretation. To visualize the results, we retrieved the top 100 most frequent tokens and displayed them in the 2D vector space. Each cluster represents elements with structural or functional similarities in the programming language. For example, we observe that 'True' and 'False' are grouped together, indicating their representation of Boolean values. Similarly, symbols like '' and '' cluster together due to their structural relevance.

To further investigate performance, we conducted additional training of our vulnerability detection model using our dataset with Word2Vec as the embedding method. For evaluating the results, we utilized the F1-score as the primary metric for analyzing model effectiveness. The findings indicate that the combination of CodeBERT and LSTM outperforms Word2Vec and LSTM across all four datasets. This suggests that the advanced contextual embeddings provided by CodeBERT significantly enhance the model's ability

doi:10.6342/NTU202401783

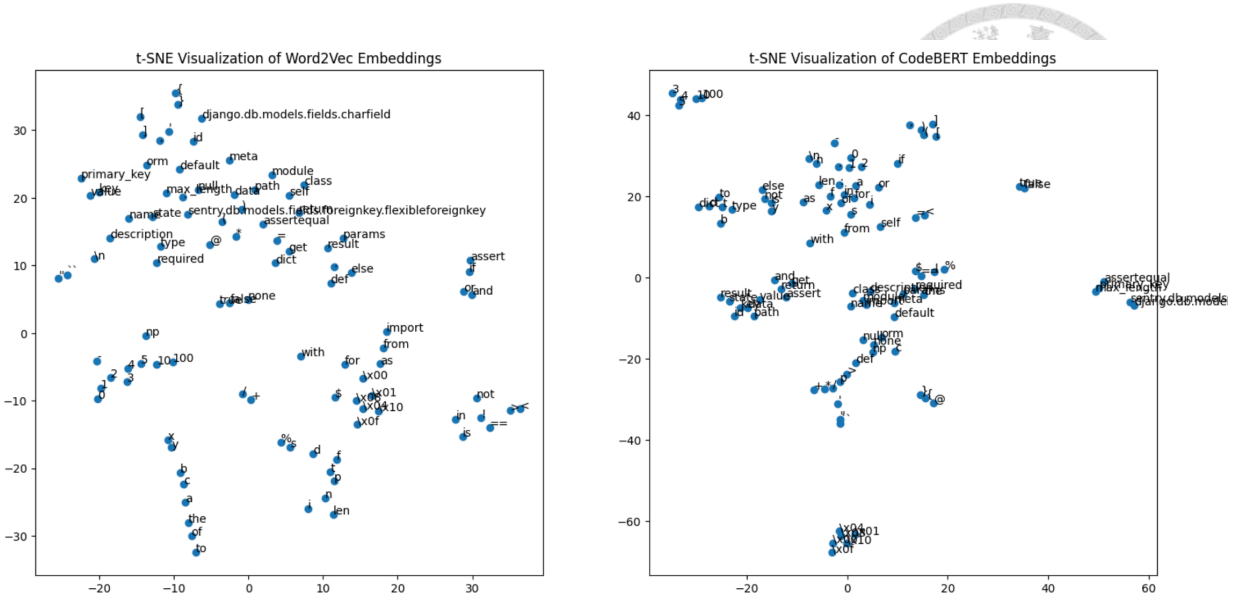


Figure 5.1: t-SNE of two embeddings method.

to detect vulnerabilities compared to traditional Word2Vec embeddings. Table 5.4 shows the results.

Model	SQL injection	XSS	XSRF	Command injection
Word2Vec+LSTM	0.80	0.85	0.88	0.89
CodeBERT+LSTM	**0.83**	**0.89**	**0.90**	**0.91**

Table 5.4: Comparison of two embedding methods with F1.

As we can see, CodeBERT and LSTM achieve overall better performance compared to Word2Vec and LSTM across all four metrics. This is likely due to CodeBERT being a bimodal model, trained to handle both natural language (NL) and programming language (PL). In modern software development, there is often a focus on writing human-readable source code, which involves using meaningful names for functions and variables, as well as writing code documentation in natural language. This results in a situation where semantic information can be extracted from the source code using techniques originally intended for NLP, such as pre-trained language representations like CodeBERT.

Fig. 5.2 shows the changes in loss and F1-score based on the number of training epochs. The blue line represents CodeBERT with LSTM, while the orange line repre-

doi:10.6342/NTU202401783

sents Word2Vec with LSTM. We observe that the Word2Vec with LSTM model converges faster, with performance improvement becoming insignificant after approximately reaching epoch 50. In fact, the training time cost of CodeBERT is ten times more than that of Word2Vec, but in terms of performance, CodeBERT ultimately prevailed.

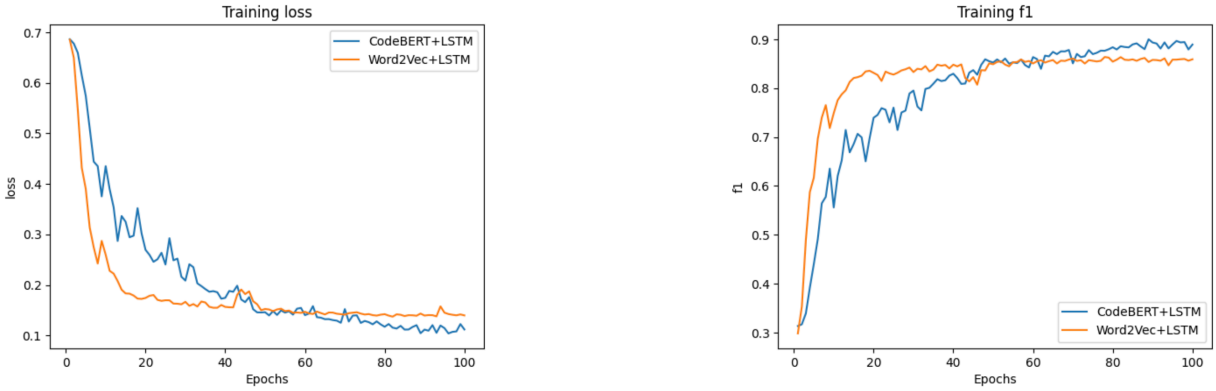


Figure 5.2: Training loss and F1-score.

5.4 Comparison with SAST tools

At the final stage of experiment, we compared our model with the existing static application security testing(SAST) tools, Bandit and Checkmarx. Since Bandit only support SQL injection and cross-site scripting(XSS) vulnerability test, in this stage we only discuss these two vulnerabilities. We collect another dataset of SQL injection and XSS, the detail of dataset illustrate in Table 5.5.

Vulnerability	Repositories	Commits	File
SQL Injection	54	74	97
XSS	49	69	94
Total	**103**	**143**	**191**

Table 5.5: Testing dataset.

doi:10.6342/NTU202401783

5.4.1 Analysis

To analyze the testing results, we visualize our work, specifically the prediction of vulnerable code. We marked the prediction outcomes as follows: true positive in blue, true negative in green, false positive in red, and false negative in pink (see Fig. 5.3).

	Marked Color
TP	
TN	
FP	
FN	

Figure 5.3: Visualization of prediction.

Fig. 5.4 and Fig. 5.5 present the scanning results of an SQL injection vulnerable sample. Both our model and Bandit identified the same part of the code. We observed that the problem lies in directly embedding the 'email' variable into the SQL query string without proper sanitization or parameterization. This makes it susceptible to manipulation by malicious users. One way to solve this issue is through parameterized queries. By using placeholders (e.g., ?, , @username) to denote where user input should be inserted, and then binding parameters to these placeholders using appropriate methods, we ensure that the input is treated as data. Consistently using parameterized queries can significantly reduce the risk of SQL injection attacks in applications. Note that this sample is different from the training dataset, the model did not see this sample before, and we successfully capture the vulnerable part of the code.

One thing worth mentioning is one of the XSS samples (see Fig. 5.6). The Markdown library in Python is used to convert text written in Markdown format into HTML.

doi:10.6342/NTU202401783

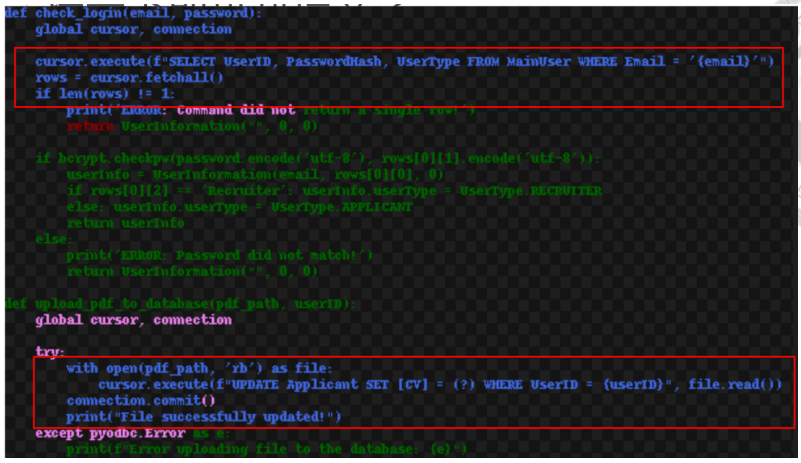


Figure 5.4: Prediction of our model.

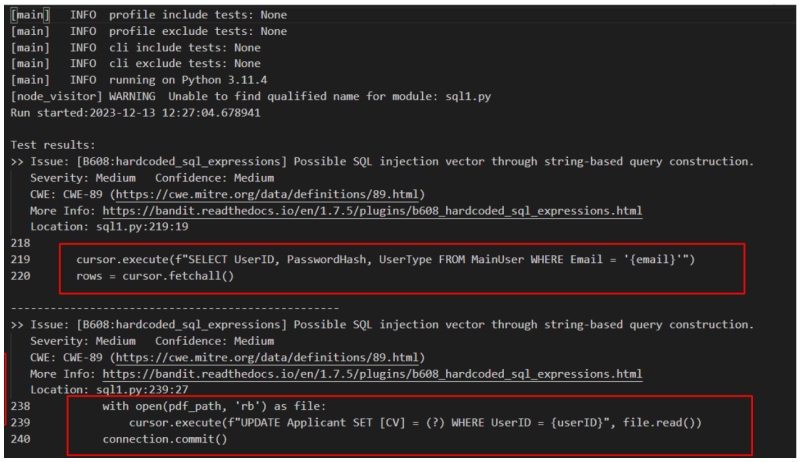


Figure 5.5: Scanning report from Bandit.

Markdown is a lightweight markup language with plain text formatting syntax, often used to create formatted text using a plain-text editor. The markdown library processes this text and translates it into corresponding HTML code, making it easier to display the formatted content on web pages [34]. This method has a parameter called safe mode. When converting Markdown syntax to HTML, if the Markdown text contains JavaScript or other HTML directives, they will also be converted to HTML. This can lead to unintended consequences because these directives will be parsed. To prevent such occurrences, you need to set the safe mode parameter to true, so it will skip those HTML tags during conversion. However, this parameter was removed after patch 2.6 because the safe mode could still be bypassed in other ways, leading to reported CVEs. Our model captures the unsafe method

doi:10.6342/NTU202401783

usage that causes XSS vulnerability, which is hard for any static analysis tool to detect. Since the deep-learning model directly learns from the cases, it can capture vulnerabilities like this.

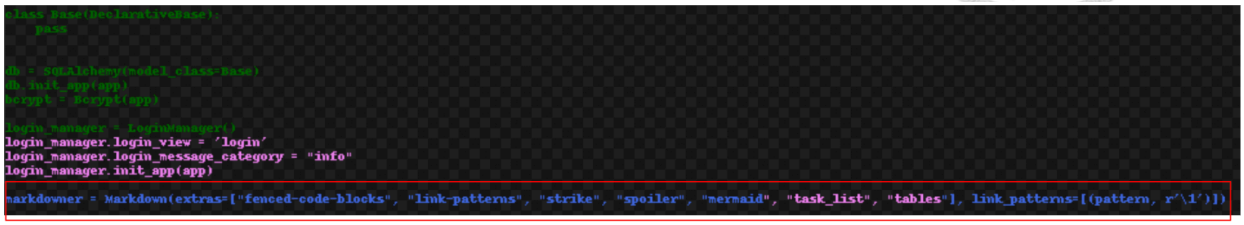


Figure 5.6: XSS sample prediction.

5.4.2 Result

Table 5.6 shows the final results. We can see that our model performs better than Bandit and slightly better than Checkmarx. For the XSS results, Bandit's performance is incredibly low. After studying the original source code, we found that there were only two predefined features of XSS vulnerability. This result demonstrates the shortcoming of static analysis tools: they heavily rely on predefined features. Checkmarx, on the other hand, as a commercial tool maintained by security experts, is still prone to some false positives (see Fig. 5.7). In this case, we can see that the query has been parameterized, yet it is still reported as vulnerable.

	SQL injection	Second order SQL injection	XSS	Stored XSS
Our model	**31**	**49**	**67**	25
Checkmarx	29	48	66	**26**
Bandit	21	40	7	2
Total samples	41	58	77	27

Table 5.6: Vulnerability found on different datasets.

doi:10.6342/NTU202401783

SQL Injection\路徑 33:

嚴重程度：	高風險
結果狀態：	校驗
線上結果	http://checkmarx.tst.gov.tw/CxWebClient/ViewerMain.aspx?scanid=1132981&projectid=105&pathid=213
結果評論	
狀態	新的
Detection Date	2/29/2024 11:02:16 AM

The application's verify_user_code method executes an SQL query with execute, at line 121 of sql/sql7.py. The application constructs this SQL query by embedding an untrusted string into the query without proper sanitization. The concatenated string is submitted to the database, where it is parsed and executed accordingly.
An attacker would be able to inject arbitrary syntax and data into the SQL query, by crafting a malicious payload and providing it via the input get; this input is then read by the verify_user_code method at line 121

of sql/sql7.py. This input then flows through the code, into a query and to the database server - without sanitization.
This may enable an SQL Injection attack.

	來源	目的地
檔案	sql/sql7.py	sql/sql7.py
行	122	126
物件	get	execute

代碼片斷

檔案名稱 sql/sql7.py

方法 def verify_user_code():

```
....
122.  code = request.args.get('code')
....
125.  values = (code,)
126.  mycursor.execute(query, values)
```

Figure 5.7: Checkmarx Scanning report of SQL injection sample.

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

Chapter 6

Conclusion

In this thesis, We first evaluate the suitable embedding method for source code embedding, the transformer based model CodeBERT perform better than Word2Vec as we expected. In order to find the optimal hyperparameter set, we leverage the simulated annealing algorithm. Next, we have designed an automatic vulnerability detection model for Python source code based on LSTM architecture. We implement our model to the real word projects and achieve a satisfying result compared to the existing SAST tool Bandit which is also a good tool of source code vulnerability scanning.

Future works would be focusing on improve the quality of training dataset, increase the quantity of dataset, and study other types of vulnerability that can be applied in our work.

doi:10.6342/NTU202401783

doi:10.6342/NTU202401783

References

[1] W. contributors, *Shellshock (software bug)*, https://en.wikipedia.org/wiki/Shellshock_(software_bug).

[2] CVE Details, *Cve details - the ultimate security vulnerability datasource*, https://www.cvedetails.com/.

[3] R. Russell, L. Kim, L. Hamilton, *et al.*, "Automated vulnerability detection in source code using deep representation learning," Dec. 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[4] Z. Li, D. Zou, S. Xu, *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, ser. NDSS 2018, Internet Society, 2018. DOI: 10.14722/ndss.2018.23158. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158.

[5] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *Journal of Systems and Software*, vol. 158, p. 110 427, 2019.

[6] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities-does experience matter?" In *2009 International Conference on Availability, Reliability and Security*, IEEE, 2009, pp. 804–810.

[7] OWASP Foundation, *Owasp testing guide v4.1*, 2021. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/v41/.

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[9] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE symposium on security and privacy*, IEEE, 2014, pp. 590–604.

doi:10.6342/NTU202401783

[10] N. Ziems and S. Wu, *Security vulnerability detection using deep learning natural language processing*, 2021. arXiv: 2105.02388.

[11] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE international conference on computer and communications (ICCC)*, IEEE, 2017, pp. 1298–1302.

[12] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *CoRR*, vol. abs/2009.07235, 2020. arXiv: 2009.07235. [Online]. Available: https://arxiv.org/abs/2009.07235.

[13] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: Vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106 809, 2022.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[15] R. Wang, S. Xu, X. Ji, Y. Tian, L. Gong, and K. Wang, "An extensive study of the effects of different deep learning models on code vulnerability detection in python code," *Automated Software Engg.*, vol. 31, no. 1, Jan. 2024, ISSN: 0928-8910. DOI: 10.1007/s10515-024-00413-4. [Online]. Available: https://doi.org/10.1007/s10515-024-00413-4.

[16] PyCQA, *Bandit*. [Online]. Available: https://github.com/PyCQA/bandit.

[17] *Checkmarx*. [Online]. Available: https://checkmarx.com/.

[18] C. Parsing, "Speech and language processing," *Power Point Slides*, 2009.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[20] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

doi:10.6342/NTU202401783

[21] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[22] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[23] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[24] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," *CoRR*, vol. abs/2001.00059, 2020. arXiv: 2001.00059. [Online]. Available: http://arxiv.org/abs/2001.00059.

[25] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, *et al.*, "A survey on software defect prediction using deep learning," *Mathematics*, vol. 9, no. 11, p. 1180, 2021.

[26] Z. Feng, D. Guo, D. Tang, *et al.*, *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155.

[27] OWASP. [Online]. Available: https://owasp.org/Top10/.

[28] N. I. of Standards and T. (NIST). [Online]. Available: https://nvd.nist.gov/.

[29] GitHub. [Online]. Available: https://github.com/advisories.

[30] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," in *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012, pp. 7–10.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[32] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *arXiv preprint arXiv:1608.02715*, 2016.

[33] F. Chollet, *Deep learning with Python*. Simon and Schuster, 2021.

[34] *Markdown library for python*, https://pypi.org/project/Markdown/.

doi:10.6342/NTU202401783