SOURCE CODE VULNERABILITY SCAN USING AI

7COM1039-0509-2022 - Advanced Computer Science Masters Project

Vineeth Gopinadhan

21059965 | University of Hertfordshire

Table of Contents

1 Introduction and Overview

1.1 Background

In the twenty-first century, software application demand increased exponentially. These applications influence all aspects of our life, from personal tasks to business operations. The software automates our daily tasks, powers the business world, drives scientific research, facilitates global communication, improves healthcare, and even provides us with entertainment. Mobile apps, desktop software, web application, and complex systems that power industries and government are now fundamental components of our society. This application's convenience, efficacy, and innovation have drastically altered our way of life, making the world more interconnected and information more accessible.

However, the widespread use of software programs has resulted in a startling rise in cyberattacks. Cybersecurity threats and breaches are more frequent and highly sophisticated. Every day, businesses, governments, and individuals face a variety of threats, like data breaches, ransomware attacks, espionage, and sabotage. According to a report by Cybersecurity Ventures, the cost of cybercrime is predicted to increase from $3 trillion in 2015 to $10.5 trillion annually by 2025 (Ene, 2023). This exponential growth illustrates the magnitude of challenges organisations and individuals face in protecting their digital assets.

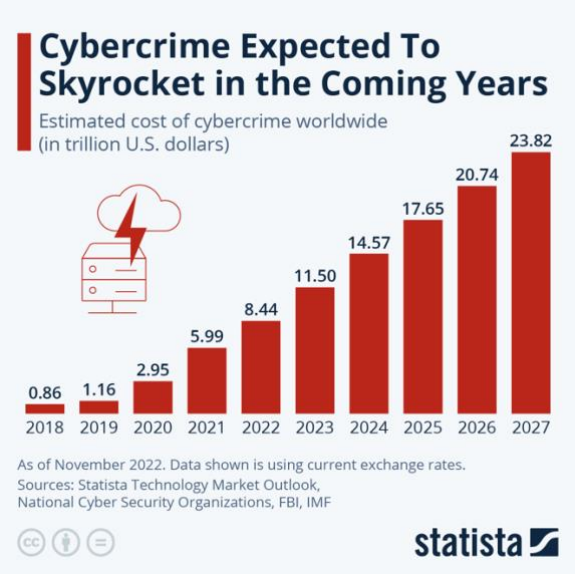

Figure 1: Estimated cost of cybercrime worldwide (Fleck, 2022)

The vulnerabilities found in the source code of software applications are the primary cause of many of these cyberattacks. A weakness or flaw in the system, which could be used to bypass security measures or issue malicious commands, is referred to as a vulnerability in this context. Numerous factors, including faulty programming, insufficient testing, the use of unsecured third-party libraries, or out-of-date components, can lead to the introduction of vulnerabilities. Such vulnerabilities have a higher chance of occurring due to the complexity and size of software applications increase. These flaws, if discovered and fixed promptly, could serve as a point of entry for attackers, potentially resulting in data loss, system damage, and catastrophic financial and reputational implications (Yang et al, 2022).

The Heartbleed (Bojanova et al., 2023), Equifax (Erickson et al., 2023), and SolarWinds (Alkhadra et al., 2021) hacks serve as examples of how traditional software vulnerability detection tools fall short in spotting sophisticated cyberattacks. These incidents highlight the requirement for more sophisticated tools that make use of Artificial Intelligence (AI) techniques to find software source code vulnerabilities. Through the analysis of enormous amounts of code, the discovery of patterns, and the identification of potential flaws that may go undetected by conventional techniques, AI can improve the precision and efficacy of vulnerability detection. To reduce the risks associated with software vulnerabilities and improve overall cybersecurity defences, AI-based source code vulnerability detection tools must be created and put into use.

1.2 Problem Statement

For effective cybersecurity measures to be implemented, source code vulnerabilities must be detected. Scanners for source code vulnerabilities based on machine learning that are currently available have shown promise in locating potential flaws. These scanners have several drawbacks and restrictions, though. First, manual feature engineering, which can be time-consuming and sometimes miss complex patterns and relationships found in source code, is a major component of traditional machine learning techniques. Furthermore, it can be difficult to find labelled training data for uncommon or newly discovered vulnerabilities, which makes it difficult for machine learning-based scanners to perform well (Fertig et al.,

2022). Third, machine learning models frequently have trouble generalising to novel and undiscovered vulnerabilities, which results in false positives or false negatives.

This research project aims to investigate the use of deep learning techniques for source code vulnerability detection to address these limitations. Deep learning models have proven to be remarkably effective at identifying complex relationships and patterns across a range of fields (Dong et al., 2021). It is proposed that the limitations of machine learning-based source code vulnerability scanners can be overcome by utilising the power of deep learning. Deep learning can automate the feature engineering procedure, allowing models to derive their learning directly from representations of the source code. Deep learning models can also successfully learn from small amounts of labelled data, which makes them more adaptable to uncommon or newly discovered vulnerabilities (Dong et al., 2021). This study aims to enhance overall cybersecurity measures by increasing the precision, efficacy, and generalizability of source code vulnerability detection using deep learning techniques.

1.3 Research Questions

1. How can CNNs improve detection accuracy by automating the feature engineering process in source code vulnerability detection, which would otherwise need human feature engineering?

2. How can CNNs efficiently learn from minimal labelled data, allowing the identification of uncommon or emergent vulnerabilities for which it is difficult to get enough labelled training data?

3. How may CNNs reduce false negatives and false positives in source code vulnerability identification by generalising effectively to new and unknown vulnerabilities?

4. How can CNN models' architecture and hyperparameters be optimised for best results in detecting source code vulnerabilities?

5. How can current software development pipelines be made to more easily accommodate the incorporation of CNN-based source code vulnerability detection systems, guaranteeing smooth adoption and integration into cybersecurity practices?

1.4 Aim & Objective

The purpose of this research project is to investigate and create a convolutional neural network (CNN)-based source code vulnerability detection system that overcomes the shortcomings of existing machine learning-based scanners. This study aims to improve the precision, effectiveness, and generalizability of source code vulnerability detection, thereby enhancing overall cybersecurity measures.

Objectives:

1. To review and assess existing research on the use of CNNs in other fields and the detection of source code vulnerabilities to gain knowledge and pinpoint best practices.

2. To investigate a suitable source code representation that can be efficiently processed by CNNs while taking into account the specific properties and structures of programming languages.

3. Creating and implementing a CNN-based source code vulnerability detection model that automates feature engineering, eliminates the need for manual feature engineering, and boosts detection precision.

4. to achieve the best performance in source code vulnerability detection by experimenting with and analysing the architecture and hyperparameters of the CNN models.

5. to assess and compare, using benchmark datasets, the performance of the CNN-based source code vulnerability detection system against conventional machine learning-based scanners and other cutting-edge techniques.

6. To offer guidelines and recommendations for the adoption and use of CNN-based source code vulnerability detection systems in practical situations, promoting improved cybersecurity measures in software development.

1.5 Deliverables

The key technical deliverables that will emerge from our extensive investigation into the creation of a source code vulnerability detection system using Convolutional Neural Networks (CNNs) are outlined in the section that follows. Increasing the precision, effectiveness, and generalizability of source code vulnerability detection is the overarching goal of these deliverables, which will also help to advance cybersecurity measures as a whole.

- CNN-based Vulnerability Detection Model: A developed and fully-functional model for identifying vulnerabilities in source code using convolutional neural networks is the main output of the research work. This model aims to increase the detection precision of current systems by automating the feature engineering process.

- Optimized Model Configurations: This deliverable will include the CNN models' optimised hyperparameters and architecture, which were discovered through a number of methodical experiments and analyses. The goal of these optimisations is to ensure that the CNN models operate at their highest levels of efficacy and efficiency.

- Benchmarking Results: A set of results comparing the CNN-based source code vulnerability detection system to conventional machine learning-based scanners and other cutting-edge techniques will be included in this deliverable. These findings will serve as a foundation for comparing and evaluating the system's efficacy.

- Deployment Guidelines and Recommendations: A set of guidelines and suggestions for using the created CNN-based source code vulnerability detection system in actual-world scenarios will make up the final technical deliverable. This document will serve as a roadmap for software developers and businesses looking to strengthen their cybersecurity defences by implementing the created system.

- Source Code and Executables: The source code and executable files for the created CNN-based vulnerability detection system will be made available. Through these, users will be able to deploy and test the system directly within their software development environments.

1.6 Software and Hardware Requirements

Python is used as the main programming language for software because of its broad support for machine learning and data processing libraries. Convolutional neural networks (CNNs) are implemented and optimised using libraries like TensorFlow or PyTorch. As it runs on top of TensorFlow, Keras, a high-level neural networks API, is also used. While Matplotlib and Seaborn offer capabilities for data visualisation, libraries like NumPy and Pandas are used for handling and manipulating data. Traditional machine-learning techniques are also used with the Scikit-learn library and will be compared to the CNN models.

In terms of hardware, a powerful computer with a multi-core processor that can handle intensive computations and sizable datasets is needed. To speed up the training and testing of CNN models, a cutting-edge graphics processing unit (GPU) is preferred. Additionally, it is recommended to have at least 16GB of RAM to ensure effective model training and data processing.

Due to its adaptability and compatibility with a variety of machine learning environments and tools, Linux is preferred as the operating system. The used software, however, is cross-platform and can be used with either Windows or macOS computers. Due to the size of source code datasets, the requirement to store intermediate results, and the need to store trained models, a solid-state drive (SSD) with a capacity of at least 1TB is used for storage. Finally, for the development, testing, and debugging of code, development environments like Pycharm and Jupyter Notebook are used. These settings make interactive coding possible and provide a useful platform for sharing and documenting the research process.

2 Progress to Date

2.1 Work Completed So Far

The research started with an overview of source code vulnerabilities as a serious problem and the need for an advanced vulnerability detection system. The research problem, its significance, and the suggested use of convolutional neural networks (CNNs) were all described in general terms in Chapter 1. It also laid the groundwork for future work by outlining the goals and potential outputs of the research. An extensive literature review was then conducted in Chapter 2. The literature that already exists on source code vulnerability detection and the use of CNNs in various fields was examined and analysed. Best practices were also discovered through this process, which will be crucial in the further development of the CNN-based vulnerability detection system.

2.2 Problems Encountered and Anticipated Challenges

The extensive range of research in both the source code vulnerability detection and the convolutional neural networks fields that was available presented a significant challenge during the literature review. It took a significant amount of time and effort to sort out the most recent and pertinent research. Furthermore, it is expected to be difficult to translate source code into a format that CNNs can use effectively. This is because programming languages have distinct characteristics and structures.

2.3 Steps Taken and Future Measures to Overcome Challenges

A systematic review methodology was used to deal with the enormous amount of literature. It was possible to effectively manage and synthesise the research by establishing clear guidelines for inclusion and exclusion and organising papers based on their applicability and contribution to the field. The idea is to investigate various methods used in related works to address the problem of source code representation for CNNs. Additionally, it is anticipated that working with subject matter experts and experimenting with various source code representation techniques will produce a workable solution.

2.4 Supporting Evidence and Documentation

The development of the research to date has been carefully documented. The research work is thoroughly introduced in Chapter 1 while the literature review is presented in Chapter 2 in great detail. These papers offer a solid framework for future research efforts and clearly show the breadth of the work done thus far.

The research is moving along quite well and on schedule, and every effort will be made to address any potential problems that may arise in the future. The design and implementation of the CNN-based source code vulnerability detection system, as well as its assessment and optimisation, will be the main topics of the following chapters.

3 Project Plan and Evaluation

3.1 Project Timeline and Major Tasks

The research project began on July 1 and is expected to be finished on August 30. The major tasks that need to be completed, including the ones that have already been completed, are listed in detail below:

1. Introduction (Completed on July 5th): The project began with the preparation of the Introduction, which defined the research's problem, purpose, objectives, and potential deliverables.

2. Literature Review (Completed on July 15th): A comprehensive literature review and analysis on source code vulnerability detection and the application of CNNs in various domains was conducted.

3. Source Code Representation (To be completed by July 25th): This task requires the creation of a method for converting source code into a format that can be processed by CNNs. The deliverable will be the finalised scheme for code representation.

4. Design and Implementation of CNN Model (To be completed by August 5th): Creating a CNN-based vulnerability detection model for source code and automating the feature engineering process. The deliverable will be a fully functional model of CNN.

5. Optimization of CNN Model (To be completed by August 15th): In this task, experimentation and analysis are used to modify the CNN models' architecture and hyperparameters. The CNN models' optimised configurations will be the output.

6. Performance Evaluation (To be completed by August 20th): Using benchmark datasets, the CNN-based detection system will be compared to conventional scanners and other cutting-edge techniques. The evaluation's findings will be the delivery.

7. Writing Final Report and Demonstration Preparation (To be completed by August 30th): In the final phases of the project, the final report is written and submitted, and preparations are made for the demonstration or presentation of the research.

3.2 Quality Assessment and Process Evaluation

The project work will be evaluated based on several factors. The first is how innovative and effective the developed CNN model is at detecting source code vulnerabilities when compared to conventional scanners. The performance evaluation results will be used as quantifiable proof of this. The second criterion is the extent to which the developed system can be generalised to different programming languages and types of vulnerabilities. The system's value and impact will increase with its breadth and adaptability.

The thoroughness and clarity of the project's final report as well as the impact of the demonstration and presentation will also be evaluated. The effectiveness of problem-solving, adherence to the schedule, and the lessons discovered throughout the course of the project will all be considered in the evaluation of the process. To ensure the project stays on track and to continuously improve the process, regular reviews and reflections will be carried out.

4 Bibliography

1. Fleck, A. (2022). Cybercrime Expected To Skyrocket in Coming Years. [online] Statista. Available at: https://www.statista.com/chart/28878/expected-cost-of-cybercrime-until-2027/ [Accessed 18th July 2023].

2. Yang, H., Yang, H., Zhang, L., & Cheng, X. (2022). Source Code Vulnerability Detection Using Vulnerability Dependency Representation Graph. In 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) (pp. 457-464). Wuhan, China. doi: 10.1109/TrustCom56396.2022.00070.

3. Bojanova, I. and Galhardo, C. E. C. (2023). "Heartbleed Revisited: Is it Just a Buffer Over-Read?" in IT Professional, vol. 25, no. 2, pp. 83-89, March-April 2023. [Online]. Available: doi: 10.1109/MITP.2023.3259119. [Accessed: July 18, 2023].

4. Erickson, S.L., Stone, M., Serdar, G. and Pfeffer, B., 2023. When Crisis Victims Are Not Customers: SCCT and the Equifax Data Breach. Journal of Managerial Issues, 35(2).

5. Alkhadra, R., Abuzaid, J., AlShammari, M., & Mohammad, N. (2021). Solar Winds Hack: In-Depth Analysis and Countermeasures. [Online]. Available at: https://doi.org/10.1109/ICCCNT51525.2021.9579611 (Accessed: July 18, 2023).

6. Fertig, A., Weigold, M. and Chen, Y., 2022. Machine Learning based quality prediction for milling processes using internal machine tool data. Advances in Industrial and Manufacturing Engineering, 4, p.100074.

7. Dong, Shi, Ping Wang, and Khushnood Abbas. "A survey on deep learning and its applications." Computer Science Review 40 (2021): 100379.

8. Fang, Y., Liu, Y., Huang, C. and Liu, L., 2020. FastEmbed: Predicting vulnerability exploitation possibility based on ensemble machine learning algorithm. Plos one, 15(2), p.e0228439.

9. Gupta, A., Sharma, S., Goyal, S., & Rashid, M. (2020). Novel XGBoost Tuned Machine Learning Model for Software Bug Prediction. In 2020 International Conference on Intelligent Engineering and Management (ICIEM) (pp. 376-380). London, UK. [Online]. Available at: doi:10.1109/ICIEM48762.2020.9160152. [Accessed: July 18, 2023].

10. Pinnamaneni, J., S, N., & Honnavalli, P. (2022), 'Identifying Vulnerabilities in Docker Image Code using ML Techniques,' in 2nd Asian Conference on Innovation in Technology (ASIANCON), Ravet, India, 2022, pp. 1-5, Available at: https://doi.org/10.1109/ASIANCON55314.2022.9908676 (Accessed: 18 July 2023).

11. Letychevskyi, O. and Hryniuk, Y., 2020. Machine Learning Methods for Improving Vulnerability Detection in Low-level Code. In: 2020 IEEE International Conference on Big Data (Big Data). Atlanta, GA, USA, pp. 5750-5752. Available at: doi:10.1109/BigData50022.2020.9377753. [Accessed: July 18, 2023].

12. Santithanmanan, K. (2022). The Detection Method for XSS Attacks on NFV by Using Machine Learning Models. In 2022 International Conference on Decision Aid Sciences and Applications (DASA) (pp. 620-623). Chiangrai, Thailand. doi:10.1109/DASA54658.2022.9765122. Available at: https://ieeexplore.ieee.org/document/9765122 (Accessed: July 18, 2023).

13. Ene, C. (2023) '10.5 Trillion Reasons Why We Need A United Response To Cyber Risk', Forbes, 22February. Available at: https://www.forbes.com/sites/forbestechcouncil/2023/02/22/105-trillion-reasons-why-we-need-a-united-response-to-cyber-risk/?sh=34d6ba963b0c (Accessed: 18 July 2023)

Appendix A: Chapter 1 - Introduction

1.1 Background

The need for software applications is skyrocketed in the twenty-first century. These applications pervade every facet of our lives, from personal tasks to professional operations. Software automates our daily chores, powers the business world, drives scientific research, facilitates global communications, enhances healthcare, and even entertains us. Mobile apps, desktop software, web applications, and complex systems powering industries and governments have become fundamental components of our societal fabric. The comfort, efficiency, and innovation provided by these applications have drastically transformed our lives, making the world more interconnected and information more accessible.

However, the widespread use of software programmes has also resulted in a remarkable rise in cyberattacks. Cybersecurity threats and breaches are more frequent and highly sophisticated. Every day, risks to organisations, governments, and people include espionage, sabotage, and ransomware assaults. The cost of cybercrime is predicted to increase from $3 trillion in 2015 to $10.5 trillion annually by 2025, according to a report by Cybersecurity Ventures (Ene, 2023). This quick rise emphasises how difficult it is for businesses and people to safeguard their digital assets.

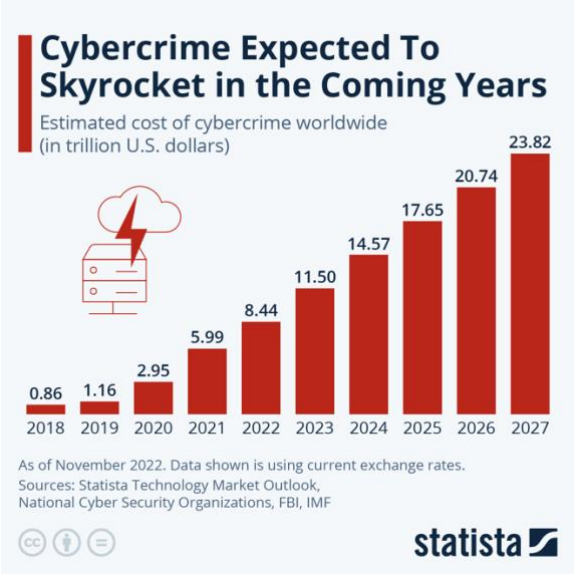

Figure 1: Estimated cost of cybercrime worldwide (Fleck , 2022)

The vulnerabilities found in the source code of software programmes are the primary cause of many of these cyberattacks. A weakness or defect in the system, which might be used to overcome security measures or issue malicious instructions, is referred to as a vulnerability in this context. Numerous factors, including faulty programming, insufficient testing, the usage of unsecured third-party libraries, or out-of-date components, may lead to the introduction of vulnerabilities. Such vulnerabilities have a higher chance of occurring due to the complexity and size of software programmes increasing. If these flaws are not found and fixed in a timely manner, they might serve as a point of entry for attackers, potentially resulting in data loss, system damage, and dire financial and reputational repercussions (Yang et al., 2022).

The consequences of source code vulnerabilities have been shown throughout time by serious cyberattacks. By taking advantage of a defect in the widely used OpenSSL library, the Heartbleed bug in 2014 jeopardised internet security. This flaw made it possible for unauthorised people to access private data, including passwords and secret codes (Bojanova et al, 2023). A vulnerability in the web development framework Apache Struts caused a data breach at Equifax in 2017 that exposed the personal information of almost 147 million customers. According to Erickson et al. (2023), one significant cause was Equifax's refusal to address the known vulnerability. The SolarWinds breach happened in 2020 when Russian hackers took advantage of a software flaw in the Orion platform of SolarWinds. They affected some 18,000 users, including US government organisations, by adding malicious malware to software upgrades (Alkhadra et al., 2021). These examples highlight the need for strong security controls and thorough software testing to guard against the exploitation of flaws in widely used software.

The Heartbleed, Equifax, and SolarWinds hacks serve as prime illustrations of how typical software vulnerability detection technologies fall short in spotting sophisticated cyberattacks. These occurrences highlight the need for more sophisticated tools that make use of Artificial Intelligence (AI) approaches to find software source code vulnerabilities. Through the analysis of enormous quantities of code, the discovery of patterns, and the identification of possible flaws that may go undetected by conventional techniques, AI may improve the precision and efficacy of vulnerability detection. To reduce the risks associated with software vulnerabilities

and improve overall cybersecurity defences, AI-based source code vulnerability detection solutions must be created and put into use.

1.2 Problem Statement

To provide effective cybersecurity measures, source code vulnerabilities must be found. Existing source code vulnerability scanners based on machine learning have shown promise in spotting possible flaws. These scanners, however, have a number of drawbacks and restrictions. First off, human feature engineering is a major component of classic machine learning algorithms, which may be time-consuming and may miss complicated patterns and correlations found in source code. Second, it may be difficult to get labelled training data for uncommon or newly discovered vulnerabilities. This makes it difficult for machine learning-based scanners to function well (Fertig et al, 2022). Third, machine learning models often experience false negatives or false positives as a result of their inability to generalise successfully to novel and undiscovered vulnerabilities.

This research project seeks to investigate the application of deep learning approaches for source code vulnerability identification to solve these constraints. Deep learning models have shown to be very effective in identifying complex correlations and patterns across a range of fields (Dong et al., 2021). It is proposed that the constraints of machine learning-based source code vulnerability scanners may be solved by using the capabilities of deep learning. Deep learning can automate the feature engineering process, allowing models to derive their learning directly from representations of the source code. Deep learning models may also successfully learn from little amounts of labelled data, which makes them more flexible to uncommon or newly discovered vulnerabilities (Dong et al., 2021). This study intends to enhance overall cybersecurity measures by increasing the precision, efficacy, and generalizability of source code vulnerability identification using deep learning methods.

1.3 Research Questions

1. How can CNNs improve detection accuracy by automating the feature engineering process in source code vulnerability detection, which would otherwise need human feature engineering?

2. How can CNNs efficiently learn from minimal labelled data, allowing the identification of uncommon or emergent vulnerabilities for which it is difficult to get enough labelled training data?

3. How may CNNs reduce false negatives and false positives in source code vulnerability identification by generalising effectively to new and unknown vulnerabilities?

4. How can CNN models' architecture and hyperparameters be optimised for best results in detecting source code vulnerabilities?

5. How can current software development pipelines be made to more easily accommodate the incorporation of CNN-based source code vulnerability detection systems, guaranteeing smooth adoption and integration into cybersecurity practices?

1.4 Aim & Objective

The aim of this research work is to investigate and develop an effective source code vulnerability detection system using convolutional neural networks (CNNs), addressing the limitations of traditional machine learning-based scanners. By leveraging the power of CNNs, this study aims to enhance the accuracy, efficiency, and generalizability of source code vulnerability detection, ultimately improving overall cybersecurity measures.

Objectives:

1. To review and assess existing research on the use of CNNs in other fields and the detection of source code vulnerabilities in order to gain knowledge and pinpoint best practices.

2. To investigate a suitable source code representation that can be efficiently processed by CNNs while taking into account the specific properties and structures of programming languages.

3. Creating and implementing a CNN-based source code vulnerability detection model that automates feature engineering, eliminates the need for manual feature engineering, and boosts detection precision.

4. to achieve the best performance in source code vulnerability detection by experimenting with and analysing the architecture and hyperparameters of the CNN models.

5. to assess and compare, using benchmark datasets, the performance of the CNN-based source code vulnerability detection system against conventional machine learning-based scanners and other cutting-edge techniques.

6. To offer guidelines and recommendations for the adoption and use of CNN-based source code vulnerability detection systems in practical situations, promoting improved cybersecurity measures in software development.

1.5 Paper Outline

Introduction

The background on the significance of source code vulnerability detection and the drawbacks of conventional machine learning-based scanners will be covered in the introduction. Convolutional neural networks (CNNs) and their potential for detecting source code vulnerabilities will be introduced. The purpose and objectives of the study will come after a clear statement of the research challenge.

Literature Review

The present approaches and strategies used in source code vulnerability detection will be covered in the literature review section. It will cover the limits of conventional machine learning-based scanners and examine their advantages and disadvantages. Additionally, it will look at how CNNs are used in various fields and assess how well they can identify source code vulnerabilities. The most effective methods and procedures for creating and improving CNN models will also be examined.

Methodology

The research strategy and actions used throughout the study will be described in the methodology section. It will go through topics including data gathering and pre-processing,

creating a suitable source code representation for CNN processing, and creating and implementing a model for CNN-based source code vulnerability detection. The assessment measures and benchmark datasets used to compare performance will also be covered, as well as techniques for training CNN models with little labelled input.

Experimental Results and Analysis

The CNN-based source code vulnerability detection system's experimental findings will be shown and discussed in this part. The system's effectiveness will be assessed, along with a comparison to conventional machine learning-based scanners and other cutting-edge techniques. We'll investigate how varied architectural and hyperparameter selections affect the effectiveness of detection. The efficiency of CNNs in identifying uncommon or newly discovered vulnerabilities will also be evaluated.

Integration and Deployment

The methods for incorporating the CNN-based source code vulnerability detection system into current software development processes will be covered in the integration and deployment section. It will cover relevant issues and difficulties that could come up during adoption and integration. Additionally, instructions and suggestions for using the CNN-based system in practical situations will be given.

Conclusion

The research effort will be summed up and its contributions will be highlighted in the conclusion section. It will summarise the main conclusions and accomplishments of the investigation, as well as the benefits of using CNNs for source code vulnerability identification. In order to pave the way for additional improvements in cybersecurity measures, the conclusion will also touch on possible future research areas in the area of source code vulnerability identification using CNNs.

Appendix B: Chapter 2 - Literature Review

2.1 Related Works

Fang et al.'s (2020) study focused on the topic of growing computer system vulnerabilities and the significance of knowing which ones are most likely to be used by attackers. They sought to quickly sort out the non-exploitable vulnerabilities and focus patching efforts on those who were most at risk since they understood that only a tiny percentage of vulnerabilities are actually exploited. By suggesting a novel exploit prediction model, fastEmbed, which combines the fastText and LightGBM algorithms, they distinguished their approach from earlier ones. This approach captures the context and meaning of language connected to vulnerabilities, unlike conventional systems that just take into account static statistical data, which is essential for anticipating exploits.

The fastEmbed model was evaluated against the most recent exploit prediction best practises. Without any time-mixing bias, it showed improved performance in both generalisation and prediction. On average, it outperformed the benchmark models by 6.283%. Specifically, the model outperformed techniques employing characteristics from the darkweb/deepweb by 33.577%, with an F1 score of 0.586 for the minority class, in predicting exploits "in the wild". The researchers reported that compared to current datasets on a platform called SecurityFocus, their model could forecast exploitability and exploitation in the wild up to 5 days in advance. Additionally, it foresaw the day-ahead distribution of Proof of Concepts (PoCs). This approach may be used with other sources of threat information, such as security blogs.

The researchers did admit that more effort is needed to increase the precision and calibre of exploits labelling and evidence of exploits in the wild. They recommended looking into more time- and coverage-efficient data sources, concentrating in particular on those exploited by hackers, such as the deep and black web. In conclusion, this work represents a significant advancement in exploit prediction by providing a more detailed strategy for selecting the most crucial fixes for system vulnerabilities.

The study by Fang et al. (2020) showed numerous improvements in exploit prediction models, but it also included a number of flaws and opportunities for development. First of all, the study concentrated on very uneven data sets, which may not provide a precise or thorough depiction of actual circumstances. Highly skewed datasets may result in models that succeed on classes that are overrepresented but fall short on classes that are underrepresented. Given the crucial significance of vulnerability exploitation, it might be disastrous to misclassify a major but uncommon occurrence.

Second, the presence of a proof of concept (PoC) in the Exploit-DB database was noted in the study. As a result, the model's accuracy might be impacted. The training data could not accurately represent the complete range of vulnerabilities. The quality of the training data determines how well a model performs, and apparent bias and incompleteness may reduce the resilience and generalizability of the model. Additionally, their model heavily relied on the crucial components of the text that dealt with vulnerabilities, which was seen to be its most crucial component. Although this method was successful in capturing the context and meaning of the text's words, it may not be adequate or thorough enough to handle complicated vulnerability prediction situations. Analysis of many technological, environmental, and human aspects is often required for vulnerability assessment; however, text alone may not adequately reflect these elements.

Additionally, Fang et al. (2020) discovered that although their model outperformed others when used with vulnerability knowledge from sources like Security Focus, it underperformed when used with the National Vulnerability Database (NVD). Due to its inconsistent performance, the model may not be consistently applicable across various intelligence sources, which would restrict its adaptability. Last but not least, in real-world scenarios, the model's capacity to forecast an exploitability and exploitation in the wild of a vulnerability on Security Focus five days earlier than current data sets may not be sufficient. A five-day forecast window would not be sufficient for the implementation of preventative measures in the case of cybersecurity attacks, which often demand urgent action and prediction.

The limits of Fang et al.'s study indicate areas that need more investigation, despite the fact that their work has improved the state-of-the-art in vulnerability exploit prediction. There is

a need for more balanced datasets, better PoC data quality, thorough feature extraction, broader application across various intelligence sources, and quicker vulnerability prediction.

Gupta et al. (2020) investigated how machine learning may be used to enhance bug identification in the early phases of the Software Development Lifecycle (SDLC). The goal of the study was to lessen the time and effort required for system and bug maintenance and to avert runtime emergencies. The researchers used the NASA-KC2, PC3, JM1, and CM1 datasets to evaluate five machine-learning models. The models were AdaBoost, XGBoost, Random Forest, Decision Tree, and Logistic Regression. Although Logistic Regression and Decision Tree fared rather well, Random Forest and AdaBoost stood out as the top performers. The greatest outcomes, nevertheless, were from the XGBoost model.

The group then made the decision to improve the already-existing XGBoost model by modifying the N_estimator, learning rate, max depth, and subsample, which they dubbed Tuned XGBoost. The study's findings showed that the Tuned XGBoost model performed better than any other model they tested. Particularly, across several datasets, the Tuned XGBoost model showed good levels of accuracy, precision, recall, and AUC. It obtained an accuracy of 97%, precision of 95%, recall of 98%, and AUC of 96% on the PC3 dataset. The model achieved 94% accuracy, 95% precision, 90% recall, and 94% AUC on the JM1 dataset. It demonstrated 95% accuracy, 97% precision, 95% recall, and 96% AUC when evaluated on the CM1 dataset. Finally, it demonstrated 94% accuracy, 93.3% precision, 95% recall, and 96% AUC on the KC2 dataset.

The study by Gupta et al. (2020) did provide outstanding outcomes for bug discovery using machine learning algorithms. There are a few issues to be aware of, however, as with any research.

First off, although the Tuned XGBoost model they suggested fared better than the other models in the majority of areas, it is far from perfect. Overfitting could result from adjusting the XGBoost model's N_estimator, learning rate, max depth, and subsample parameters. When a model gets extremely complicated and excels on training data but fails to generalise to fresh, untried data, it is said to be overfit. Although Gupta and his colleagues claimed better

results, it's not obvious whether the model can continue to perform at this high level with other datasets or in practical applications.

Second, to solve the class imbalance problem in the datasets, the study largely depends on the Synthetic Minority Over-sampling Technique (SMOTE). SMOTE may enhance model performance, but it also generates synthetic data that might not always accurately represent actual circumstances. This suggests that the models' excellent performance may not totally apply to real-world circumstances and may be partially explained by these synthetic findings.

The researchers' method for feature scaling at the data pre-processing stage could possibly have drawbacks. A method for standardising the range of independent variables or features in data is called feature scaling. The assumption that all characteristics are equally relevant is not necessarily true, despite the fact that this is a prevalent practice in machine learning to improve models' performance. In fact, certain characteristics could be more important than others for forecasting software defects. Their approach doesn't seem to account for this possible variance.

As a result, even if Gupta et al.'s (2020) study shows a positive development in early bug discovery, there are important limits to take into account, especially in the area of source code vulnerability detection. Although their Tuned XGBoost model performed well in trials, there is a chance that overfitting may cause it to perform poorly in practical applications. Additionally, the model's strong reliance on the SMOTE approach can cause an imbalance between artificial observations and real-world data, which might result in false positives or false negatives when detecting vulnerabilities. Their research's widespread use of feature scaling might obscure critical vulnerability indications and risk missing vital elements of bug discovery.

A machine learning model was created by Pinnamaneni et al. (2022) study to discover insecure Python libraries in Docker container images after they analysed the security vulnerabilities in these images. Due to their characteristics, Docker container images are favoured over virtual machines since they include everything required to operate an

application. The researchers chose to concentrate on static code analysis since they may potentially pose security problems.

To classify the code as either susceptible or not, the team employed a variety of supervised classification machine learning methods, including Linear Regression, Decision Trees, Naive Bayes, K-Nearest Neighbours (KNN), Support Vector Machine (SVM), Random Forest, Gradient Boost, and XGBoost. According to the research, the algorithms for Decision Tree, Random Forest, Gradient Boost, and XGBoost all obtained 100% accuracy and precision.

They employed soft voting between the Decision Tree, Random Forest, and Gradient Boost algorithms to build a model that can manage any size database. They next used Streamlit to design a user interface that leverages the developed model to determine if new code is susceptible. This interface recognises the programming language of the code, and if Python is used, it determines if the code is susceptible and names the libraries that are responsible.

Their study has helped develop a useful technique for scanning and locating Python code vulnerabilities in Docker container images. The database should be expanded to include other programming languages, such as C, C++, Java, and Go, and further vulnerabilities should also be added, they suggest.

The study of Pinnamaneni et al. (2022) is a significant step towards securing Docker container images, although it does have certain limitations, especially in terms of how it approaches source code vulnerability identification. First off, their model lacks adaptability since it was trained just to find vulnerabilities in Python libraries. As a result, when used on code created in other languages, its effectiveness drastically drops. As a result of its Python focus, it cannot be used to scan the source code of programmes written in other well-known programming languages, such as C, C++, Java, or Go.

Second, while reaching purportedly 100 percent accuracy and precision, the machine learning algorithms utilised may be in danger of overfitting, which might result in a high percentage of false positives or negatives when applied to fresh, untested data. When the model is exposed to data including new kinds of vulnerabilities or libraries that weren't included in the training data, overfitting may also make the model perform badly. The report also ignores additional dangers including DOS attacks, cross-site scripting, memory corruption, and overflow

vulnerabilities since it only focuses on keylogging vulnerabilities. Users may be exposed to a broad variety of possible threats as a result of this tool's lack of comprehensiveness.

Last but not least, the model's dependence on libraries to assess vulnerability may leave out other crucial aspects of the code, which may result in vulnerabilities that do not directly touch these libraries. As a result, it can fail to notice certain vulnerabilities.

To sum up, there are certain drawbacks to the study (Pinnamaneni et al. 2022) on the security of Docker container images and the identification of Python library vulnerabilities. The method's ability to scan code written in other languages is limited by its Python-specific nature. Despite claiming great accuracy, the used machine learning models may overfit the data and result in incorrect classifications on previously unknown data. Additionally, while keylogging vulnerabilities are the emphasis, other attacks may go unnoticed, possibly putting users at danger. Last but not least, by depending entirely on libraries to forecast vulnerability, the model may miss other important components of code vulnerabilities unrelated to these libraries.

In their study, Letychevskyi et al. (2020) used machine learning (ML) to improve the identification of flaws in low-level code. To find possible weaknesses, they employed symbolic modelling to depict computer programmes as a directed graph. By fusing symbolic modelling and machine learning, the attainable route-generating process was intended to be improved. The group created a particular method for producing training data via their trials. They made use of numerous node-embedding techniques, including LLE, SPE, node2vec, DeepWalk, and artificial neural networks (ANN) and support vector machines (SVM) for classification.

The results showed that an accuracy rate of 86% was attained using the ANN approach in conjunction with node2vec node embedding. Tests utilising node-embedding techniques based on matrix factorization produced less favourable results, which may be related to the sparse nature of control flow graphs (each node may have up to two successors).

The researchers came to the conclusion that symbolic modelling methods might be greatly complemented by an ML-based strategy for determining the shortest route. They emphasised that the precision attained demonstrated the potential of their original strategy. In addition, they suggested expanding the set's size and variety, refining the node-embedding algorithm's

hyperparameters, and developing a specific classifier to assess nodes' connectivity because their previous work assumed that connections between nodes always existed even though this may not always be the case.

Despite the study's positive results, there are several possible drawbacks. First off, although control flow graphs are often sparse networks, the research relied on a node-embedding approach that may not perform well for such graphs. Its wide application may be constrained by the research's inferior performance with matrix factorization-based node-embedding techniques. In real-world applications, the link between nodes doesn't always exist, according to the methodology's second assumption. This can result in mistakes or omissions during the vulnerability identification procedure.

A variety of ML classification techniques were also tried in the research, and it was discovered that the Artificial Neural Network (ANN) approach using node2vec node embedding obtained 86% accuracy, which, although excellent, also highlighted possible drawbacks. The success of the ANN in this situation demonstrates its potential for source code vulnerability scanning.

Artificial neural networks are promise for identifying vulnerabilities because of a number of benefits. The capacity to learn and model non-linear and complicated connections, which is highly helpful for comprehending the interactions inside source code, makes them outstanding at capturing complex relationships in data and coping with large dimensionality. Additionally, ANNs are very tolerant to noisy input, which is a characteristic of big code bases often.

In conclusion, although showing that machine learning may be used to discover code vulnerabilities, the Letychevskyi et al. (2020) research has certain drawbacks. The complexity of actual software systems may not be accurately captured by the assumptions and methodologies utilised, and it is possible that certain vulnerabilities may go unnoticed. Nevertheless, the study's performance of the artificial neural network points to a potential direction for source code vulnerability scanning research and development in the future. Future research should look at enlarging and diversifying the data set, enhancing the hyperparameter optimisation of the node-embedding method, and creating specific classifiers to evaluate node connectedness.

Santithanmanan et al. (2022) conducted a research study that concentrated on Cross-Site Scripting (XSS) assaults, a widespread issue in online applications, and especially targeted Cisco Enterprise's Network Function Virtualization Infrastructure Software (NFVIS). In these XSS attacks, malicious code (typically JavaScript) is injected and then distributed to other users through a URL. Due to the possibility of sensitive user data being stolen, the hazard is very serious.

The researchers employed machine learning methods to pinpoint the URLs that could be responsible for these assaults. The k-Nearest Neighbours (k-NN), Decision Tree, Support Vector Machine (SVM), and Gaussian Naive Bayes machine learning techniques were used. On a dataset that was divided between training and testing portions in a ratio of 70:30, these models were trained and assessed. The k-NN model was the most accurate of the four evaluated approaches, according to the study's findings. In only 0.1633 seconds, it was able to determine if a URL was hazardous or benign, achieving an extremely high accuracy rate of 99.6%. It is a promising tool for the quick identification and defence against XSS assaults because of its precision and speed. As with every study, there are, nevertheless, certain gaps or opportunities for improvement.

The study's chosen machine learning models, such as k-NN, Decision Tree, SVM, and Gaussian Naive Bayes, all have inherent drawbacks. Although the k-NN model's excellent accuracy of 99.6% was obtained, it may not perform as well when dealing with big data sets or when the input dimensions are high. Additionally, it presumes that all traits are equally important, which may not always be the case. Furthermore, the use of a 70:30 split to divide the dataset into training and testing sets might result in overfitting. A typical issue in machine learning is overfitting, when a model performs well on training data but struggles to generalise to new data. This might imply that the models would perform less well in a practical application than they would in a lab setting.

Additionally, even if the research advances the automated classification of XSS assaults, it only pays attention to two properties: URL and JavaScript. As XSS attacks may be implemented through various techniques, this may create detection system gaps. Additionally, the paper avoids addressing the possibility of false positives or negatives, which might pose serious problems for machine learning-based vulnerability identification.

While a high false negative rate might imply overlooking important vulnerabilities, a high false positive rate could result in actions being made that are unnecessary.

In conclusion, although making important advancements in XSS vulnerability identification, Santithanmanan et al.'s study (2022) has certain drawbacks. The 70:30 dataset split runs the danger of overfitting since the machine learning algorithms utilised, especially the k-NN model, have intrinsic limits when handling huge data sets or high input dimensions. The research may have overlooked other possible attack paths due to its emphasis on URL and JavaScript as the only criteria for identifying XSS assaults. Furthermore, a crucial issue in machine learning-based vulnerability identification, the likelihood of false positives or negatives, was not addressed in the study. Due to these drawbacks, it is possible that more thorough methods including more features and various machine learning algorithms would be required for more accurate and reliable vulnerability identification, even though the work makes a vital contribution.

2.1.1 Conclusion

It is clear from the thorough literature study that a variety of machine learning models have been effectively used in this area, each providing a distinctive viewpoint and solution. Nevertheless, it is evident that these studies have inherent drawbacks, including overfitting, data imbalance, a limited scope, and false positive and negative concerns.

Nevertheless, these difficulties provide a strong justification for the use of deep learning, and more particularly, Convolutional Neural Network (CNN) algorithms, in the identification of source code vulnerabilities. Because deep learning can learn from vast, varied data sets and has inherent robustness against overfitting, it has shown promise in tackling issues like overfitting and data imbalance. It is more likely to find vulnerabilities that are not simply limited to certain code qualities because CNNs' high dimensionality processing capacity enables them to discover complicated patterns in source code that other models may overlook.

Additionally, the inbuilt capability of CNNs to maintain the spatial hierarchy in data may eventually result in a more comprehensive approach to vulnerability identification, resolving the problem of the research' restricted emphasis. Last but not least, CNNs may successfully

reduce the percentage of false positives and negatives, hence improving the dependability of vulnerability detection systems. The adoption of deep learning, specifically CNN algorithms, in source code vulnerability detection presents a promising avenue to overcome current limitations and enhance the reliability and robustness of vulnerability detection systems, even though previous research has made significant advances in the field.