

Application of Advanced AI Methods for Precise Vulnerability Detection

Amirreza Bagheri

Department of Software Engineering
University of Szeged

Szeged, 2025

Supervisor:

Dr. Péter Hegedűs

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED



University of Szeged

PhD School in Computer Science

Preface

My fascination with computers began at the age of seven, a time when computers were bulky and the internet was accessed through dial-up connections. Despite the limitations of these early machines, they captured my imagination and sparked a lifelong passion for technology. I started by writing simple scripts for fun, which was my first step into the world of programming. This early exposure to computers not only entertained me but also laid the groundwork for my future studies and career. As I grew older, my hobby turned into a more serious academic pursuit. I studied computer engineering in Iran, where I earned my master's degree. During my university years, I gained practical experience by working in various tech-related jobs. I assembled PCs, set up server racks, and managed network installations. These hands-on experiences were crucial; they helped me understand the practical aspects of my studies and prepared me for the challenges of the tech industry. My academic journey led me to Szeged University in Hungary for my PhD studies under the guidance of Dr. Peter Hegedus. I am deeply grateful for his mentorship over the past four years. His support and guidance were invaluable as I navigated through the complexities of advanced computer science. This thesis is not just a reflection of my academic efforts but also a testament to the supportive and collaborative environment at Szeged University. I extend my heartfelt thanks to all who have supported me on this journey. In addition to the academic and professional support I have received, I must express my deepest gratitude to my parents, who have been my pillars of strength and encouragement throughout my life. Their unwavering support and belief in my abilities have been fundamental to my success. I am also thankful for the camaraderie and support of my friends and family, who have provided both motivation and relief from the rigors of academic life. The journey of a researcher can often be solitary, involving long hours spent coding and testing in isolation. The thrill of seeing a piece of code finally work after numerous attempts, however, brings immense satisfaction and joy. This joy is a reminder of why I chose this path. The pursuit of science is akin to building a house, where each researcher contributes a brick to the structure of knowledge. In my own journey, each line of code and every successful experiment has felt like placing a brick in the vast edifice of computer science. This process, while often challenging and meticulous, is incredibly rewarding. It is a privilege to contribute to the collective endeavor of expanding human understanding and capability through technology. As I reflect on my PhD journey, I am reminded of the larger purpose of our work and the collaborative effort it represents.

Amirreza Bagheri, 2025

Contents

Preface	iii
1 Introduction	1
1.1 Structure of the Dissertation	5
1.2 Summary of the Results	6
2 Background	9
2.1 Traditional Methods	9
2.1.1 Static Analysis	9
2.1.2 Dynamic and Hybrid Analysis	10
2.2 Machine Learning Methods	11
2.2.1 Deep Learning and Neural Networks	11
2.2.2 Recurrent Neural Networks	12
2.2.3 Long Short Term Memory Networks	13
2.2.4 Large Language Model	14
2.3 Representing Code	15
2.3.1 Abstract Syntax Trees	15
2.3.2 Control Flow Graph	16
2.3.3 Data Flow Graph	16
2.3.4 Code as Natural Text	17
2.4 Embedding Code in a Numerical Vector	17
2.4.1 Word2Vec Embedding	17
2.4.2 FastText	18
2.4.3 CodeBERT	19
2.5 Self-Attention Based Methods	19
2.5.1 Transformer	19
2.5.2 Conformer	20
2.6 Mining Software Repositories	21
2.6.1 Approaches for Mining Software Repositories	21
2.6.2 Tools for Mining Software Repositories	21
2.6.3 Mining Github	22
2.7 Python Vulnerabilities	22
2.7.1 SQL Injection	22
2.7.2 Command Injection	23
2.7.3 Remote Code Execution	23
2.7.4 Cross-Site Scripting	24
2.7.5 Cross-Site Request Forgery	24
2.7.6 Directory Traversal / Path Disclosure	25

3	Related Work	27
3.1	Traditional Approaches	27
3.1.1	Vulnerability Prediction Based on Software Metrics	27
3.1.2	Anomaly Detection Methods for Finding Vulnerabilities	28
3.1.3	Vulnerable Code Pattern and Similarity Analysis	29
3.2	Alternative Approaches	30
3.2.1	Machine Learning-Based Approaches	30
3.2.2	Deep Learning for Vulnerability Prediction	31
3.2.3	Long Short Term Memory Networks	33
3.2.4	Large Language Model-Based Approaches	33
4	Sequential Modeling with LSTM and Code Embeddings	37
4.1	Choosing a Programming Language	37
4.2	Choosing Data Source	38
4.3	Choosing Vulnerabilities	38
4.4	Labeling Code Segments	39
4.5	Representation of Source Code	39
4.5.1	Choosing a Representation	40
4.5.2	Choosing Granularity	40
4.6	Collecting the Data	40
4.6.1	Scraping GitHub	41
4.6.2	Filtering the Results	42
4.6.3	First Misguided Attempt: Using Only Diffs	43
4.6.4	Downloading the Dataset	43
4.6.5	Flaws in the Data	45
4.6.6	Filtering Data	46
4.6.7	Second Misguided Attempt: Subtle Errors in Creating the Dataset	46
4.7	Processing the Data	47
4.8	Word2Vec and FastText Embeddings	49
4.9	The Final Dataset	52
4.10	Baseline Model Architecture	52
4.10.1	Preprocessing the Code	52
4.11	First Attempt with LSTM	53
4.11.1	Results of the Default Metrics	53
4.11.2	Hyperparameters of the Embedding Layers	54
4.11.3	Parameters in Creating the Dataset	56
4.11.4	Hyperparameters and Performance of the LSTM Model	57
4.11.5	Performance for Subsets of Vulnerabilities	61
4.12	Limitations and Threats to Validity	62
4.13	Summary	65
5	Conformer-LLM Integration for Block-Level Python Vulnerability Detection	67
5.1	Structural Information	67
5.1.1	Code Sequence Embedding (CSE)	68
5.1.2	Conformer	68
5.2	Building Graph	70
5.3	Building CSE	71
5.4	Network Implementation	72

5.5	Conformer Model	72
5.6	Training the Network	73
5.6.1	LSTM Setup	74
5.6.2	Preparing the Data for Classification	75
5.6.3	Training the LSTM	75
5.6.4	Conformer Setup	75
5.6.5	Data Preparation for Conformers	76
5.6.6	Training and Tuning Conformers	78
5.6.7	Large Language Models (LLMs) Setup	79
5.6.8	Fine-Tuning UniXcoder and Training	79
5.6.9	Hybrid Model Integration	80
5.7	Selecting the Machine Learning Model	81
5.8	Iterative Training Phases	82
5.9	Hyperparameter Tuning	82
5.9.1	LSTM Tuning	82
5.9.2	Conformer Tuning	83
5.9.3	LLM Tuning	83
5.10	Fusion Mechanism	83
5.11	Second Attempt with Conformer and LLM	84
5.11.1	Hybrid Model Overview	84
5.11.2	Hyperparameters of the Hybrid Model	84
5.11.3	Preprocessing Hyperparameters for AST, CFG, DFG	87
5.11.4	Hyperparameters for LLM	88
5.11.5	Performance for Subsets of Vulnerabilities	89
5.11.6	Ablation Study of Hybrid Components	89
5.11.7	Comparison with Other Methods	91
5.12	Performance Across Vulnerability Subsets	93
5.12.1	SQL Injection	93
5.12.2	Cross-Site Scripting	94
5.12.3	Command Injection	95
5.12.4	Cross-site Request Forgery	95
5.12.5	Remote Code Execution	97
5.12.6	Path Disclosure	98
5.13	Comparison with Other Works	99
5.14	Limitations and Threats to Validity	99
5.15	Summary	101
6	Conclusion	103
	Appendices	104
A	Summary in English	105
A.1	Summary in Hungarian	108
	Bibliography	108

List of Tables

1.1	Thesis contributions and supporting publications	7
3.1	Different Approaches	31
4.1	Similarity Scores for Words	51
4.2	Vulnerability Dataset	52
4.3	Baseline LSTM Performance Across Embeddings (SQL Injection Dataset)	54
4.4	F1 Scores Across Vector Dimensionalities	55
4.5	F1 Scores Across Iterations With/Without String Replacement	56
4.6	F1 Scores Across Different Iterations for min_count 10 and 5000	56
4.7	Performance Metrics for Word2Vec	57
4.8	Performance Metrics for FastText	57
4.9	Performance Metrics for CodeBERT	57
4.10	F1 Scores Across Different Numbers of Neurons	58
4.11	F1 Scores Across Different Batch Sizes	59
4.12	F1 Scores Across Different Optimizers (10 Epochs)	59
4.13	F1 Scores Across Top Optimizers (50 Epochs)	59
4.14	F1 Scores Across Dropout Rates	60
4.15	F1 Scores Across Different Epochs	60
4.16	LSTM + word2vec results for each vulnerability category.	61
4.17	LSTM + fastText results for each vulnerability category.	62
4.18	LSTM + CodeBERT results for each vulnerability category.	62
5.1	Comprehensive Training Network Setup Configurations	78
5.2	Number of Conformer Blocks Evaluation	85
5.3	Attention Heads Evaluation	86
5.4	Convolutional Kernel Size Evaluation	86
5.5	Embedding Dimension Evaluation	86
5.6	Feed-Forward Dimension Evaluation	87
5.7	Graph Embedding Dimension Evaluation	88
5.8	Sequence Length Evaluation	88
5.9	Learning Rate Evaluation for UniXcoder Fine-Tuning	89
5.10	Class Weight Evaluation for UniXcoder Fine-Tuning	89
5.11	LSTM + Conformers results for each vulnerability category.	90
5.12	LSTM + Conformers + LLM results for each vulnerability category. . .	90
5.13	Ablation Study	91
5.14	Our model comparison to other methods with same database	91
5.15	Our model comparison to other methods with same data-mining method	92

List of Figures

2.1	Example illustration of a neural network	12
2.2	Recurrent neural network. Left showing neurons with example connections, right simplified visuals for sequential data	13
2.3	LSTM network	14
2.4	The word2vec embedding	18
4.1	Retrieving the snippet in the state before and after the commit from a git diff, old vulnerable version in red, new version in green	44
4.2	Process of splitting the whole code with vulnerable (red) and non-vulnerable (green) parts in snippets for the dataset. Notice that the splitting happens first, and then vulnerable and not vulnerable are separated.	47
4.3	Process of splitting the whole code with vulnerable (red) and non-vulnerable (green) parts in snippets for the dataset	48
4.4	Process of splitting the whole code with vulnerable (red) and non-vulnerable (green) parts in snippets for the dataset	48
4.5	Processing the data from code snippet	49
4.6	LSTM baseline model	53
4.7	F1 Score across different dimensionalities.	54
4.8	F1 Score across iterations or context size (128-256)	54
4.9	F1 Score across different numbers of neurons	58
4.10	F1 Score across different batch sizes	58
4.11	F1 Score across different epochs	60
4.12	F1 Score across different dropout rates	60
5.1	Conformer Segments	68
5.2	The steps of creating AST , CFG and DFG	71
5.3	Conformer model	73
5.4	The steps of creating the model and order in which the hyperparameters come into play	74
5.5	Hybrid segments of our combined model	80
5.6	F1 Score for Graph-embedding-dimension	88
5.7	F1 Score for Sequence Length	88
5.8	Vulnerable code commit example. (SQL injection)	93
5.9	Detection of vulnerability	93
5.10	Vulnerable code commit example. (XSS)	94
5.11	Detection of vulnerability (XSS)	94
5.12	Vulnerable code commit example. (Command injection)	95
5.13	Detection of vulnerability (Command injection)	95
5.14	Vulnerable code commit example (XSRF)	96

5.15	Detection of vulnerability (XSRF)	96
5.16	Vulnerable code commit example (Remote Code Execution)	97
5.17	Detection of vulnerability (Remote Code Execution)	97
5.18	Vulnerable code commit example (Path disclosure)	98
5.19	Detection of vulnerability (Path disclosure)	98

To my parents,

1

Introduction

Computer programs are essential for the functioning of modern life. Software is vital for various sectors of society, such as healthcare, energy, transportation, public safety, education, and entertainment. Developing safe, dependable, and secure software is a complex and challenging endeavor. Software vulnerabilities can result from oversights and errors committed by software architects and engineers, leading to serious repercussions. An exploit such as the ransomware WannaCry [3] can take down hospitals and transportation networks, resulting in hundreds of millions of dollars in damage. A minor imperfection in the code, whether it spans a few lines or just one, might create a significant vulnerability, making the system susceptible to attacks. The Heartbleed [24] problem, a critical vulnerability in the OpenSSL [24] cryptographic library that impacted billions of internet users, might have been avoided by adding two additional lines of code. Cyberattacks pose a growing threat to governments, corporations, and consumers, resulting in an estimated annual cost of 400 billion dollars. The Common Vulnerabilities and Exposures (CVE) [16] database contained over 125,000 entries in January 2025. Despite the availability of secure application development guidelines, many developers fail to follow them, leading to the introduction of vulnerabilities. Additionally, outdated software is particularly susceptible to exploitation.

Code inspection is a crucial procedure for identifying vulnerabilities and is essential before addressing them. Secure code can quickly become vulnerable despite initial protections. Any modification to the code in a project has the potential to change the attack surface or introduce a security risk. Hence, consistent and meticulous code reviews are essential to identify any such imperfection. Examining code manually is a laborious and time-consuming task that demands significant expertise in security due to the rising complexity and interconnectivity of current software systems, where issues may be hidden in seemingly harmless code portions. Vulnerabilities can arise from breaching unstated, implicit programming guidelines that are challenging to monitor. Software testers must understand the programming language, software structure, and harmful tactics to effectively identify vulnerabilities. They should be able to think like an attacker. Furthermore, numerous security flaws may not impact the standard operations of a system, allowing them to remain unnoticed for an extended period. Hidden

vulnerabilities can have catastrophic consequences as they are often identified long after the problem was introduced, allowing attackers sufficient time to exploit the vulnerability. Vulnerable code can spread rapidly between projects due to the widespread use of open-source software and practices like code reuse through platforms like GitHub forks. Despite spending an extensive amount of time searching for vulnerabilities, defenders can never be completely certain they have identified all potential threats. In contrast, an attacker only needs to discover one exploitable vulnerability to inflict damage, such as crashing a program or exposing sensitive information. Developers find it challenging to remain at the forefront of their field. Identifying vulnerabilities relies heavily on individual skill due to the lack of a comprehensive systematic approach that covers all potential faults. The quality of defined features in a vulnerability detection system depends on the individual creating them. To enhance the outcome, it is necessary to involve multiple experts and combine their results, which would increase the manpower needed. It is preferable to minimize the requirement for human work.

Various technologies are currently utilized to assist developers by aiding in prioritization, testing, and minimizing the time required for tasks. The prevailing method has primarily been a formal one, focusing on modeling programming languages using mathematical structures and verifying specific aspects, such as code coverage and data flows. Symbolic execution involves replacing input data with symbolic values and analyzing how they change throughout the program's control flow. Static analysis tools examine source code using a rule-based method without executing it. Commonly used tools are FindBugs [50] for Java, Splint and Flawfinder [77] for C, and PyT [52] (Python taint) for Python. They are crucial but not adequate to address the widespread existence of vulnerabilities. Many modern tools that utilize static analysis are plagued by a high rate of false positives, impeding their widespread use and causing various issues as outlined below. Furthermore, these tools often require criteria to be established by experts, resulting in potential delays in keeping up with the latest advancements.

A novel methodology is developing utilizing data analysis and machine learning techniques. The aim is to create a vulnerability detection system that does not depend on subjective expert views and avoids high rates of false negatives or false positives. The main concept is to extract patterns from extensive datasets by utilizing a machine learning algorithm to identify vulnerability characteristics and categorize code accordingly. Vulnerability detection can be automated and implemented early in the software development process, leading to reduced expenses associated with identifying and correcting problems. Machine learning models can be updated with new data if a new type of vulnerability or attack is discovered. This method decreases dependence on individual subjective experiences and provides a more objective and universal way to define characteristics. Ultimately, it has the potential for significant scalability. Regrettably, the general adoption of automatic vulnerability discovery has not been achieved. An IBM-funded study assessed the rate of adoption and impact of automation, specifically machine learning and artificial intelligence, in identifying cyber-attacks. They concluded that utilizing different automation strategies is highly effective in preventing exploits. However, approximately 77% of firms utilize automation to a limited extent or not at all, indicating a substantial opportunity for enhancement. The ideal approach is to automate the process of identifying vulnerabilities in a manner that is more precise and considerably quicker than manual code examination. The system would not rely on subjective qualities that are manually specified but instead learn features from actual code and adjust automatically to new difficulties. This program would assist

human specialists by automating the most time-consuming and error-prone activities involved in identifying vulnerabilities. A tool with a low probability of false positives and false negatives would greatly aid developers by identifying potential vulnerabilities in the source code.

Machine learning techniques are now less commonly utilized than static analysis and other classical methods. Research has focused on machine learning for identifying vulnerability features. Many studies employ synthetic code examples, tiny datasets, or are limited to certain projects. Several prevalent programming languages have not yet garnered any notice. Moreover, numerous suggested methods solely categorize entire files, which is simpler to accomplish but less beneficial for developers. Various methods have been used to describe code and develop a model. However, approaches such as bag-of-words models and basic classification algorithms are insufficient in capturing the sequential flow and semantic organization of source code. Therefore, there is still a significant amount of work remaining.

In this thesis, we attempt to explore advanced AI techniques for precise vulnerability detection and having the best option provide a proof-of-concept tool, a tool that utilizes a deep neural network to acquire vulnerability characteristics from a substantial codebase extracted from GitHub. our model specializes in the Python programming language and exclusively learns from authentic code to ensure practical application in real-world projects. The tool operates on the source code directly, utilizing embeddings like CodeBERT [25] to encode code into numerical vector representations. our model employs a detailed approach that examines individual code tokens to classify specific areas of the code. A Long Short-Term Memory (LSTM) network is utilized to model vulnerabilities, as it can employ an internal 'memory' to grasp the semantic context of code tokens. our model is a proof of concept demonstrating a promising method for a vulnerability detection tool utilizing deep learning. Its relevance to coding is shown through several instances.

Following the adoption of LSTM to model sequential dependencies in code tokens, our model integrates two advanced mechanisms—Conformers and Large Language Models (LLMs)—to further enhance its capability to detect vulnerabilities at a block level. These components address limitations in traditional deep learning approaches, such as the inability to effectively fuse structural and semantic information or capture both local and global code contexts simultaneously.

The Conformer, originally proposed by Gulati et al. [33], combines the strengths of convolutional neural networks (CNNs) and self-attention mechanisms from Transformers. While LSTMs excel at modeling sequential data with an internal memory, they struggle with long-range dependencies and lack the parallel processing efficiency of Transformer-based models. The Conformer overcomes these by incorporating a convolutional module to capture local, position-wise features and a self-attention module to model global, content-based interactions. This dual approach is particularly suited to Python vulnerability detection, where vulnerabilities often manifest through subtle interactions between local syntax and broader program logic. In our model, the Conformer processes tokenized code sequences to extract intricate feature patterns, enabling a more nuanced understanding of block-level vulnerabilities compared to the sequential focus of LSTM.

Complementing the Conformer, our model leverages a pre-trained Large Language Model to extract rich semantic information from code snippets. LLMs, such as UniX-coder [35] based on Transformer architectures, are pre-trained on vast corpora of code

and natural language, endowing them with an inherent ability to understand programming syntax, semantics, and context. Unlike traditional embeddings like Word2Vec [15], which our model uses initially, LLMs produce contextual token representations that adapt to the surrounding code, capturing meaning beyond static word associations. For instance, an LLM can discern that a variable named `input-str` in a Python snippet likely carries user input, potentially flagging it as a risk for injection vulnerabilities. In our model, the LLM is fine-tuned on a curated dataset of Python vulnerabilities, enhancing its ability to identify semantic patterns indicative of security flaws. This pre-training reduces the training burden on downstream models like the Conformer and LSTM, improving efficiency and generalization.

Together, the Conformer and LLM elevate our model beyond LSTM’s capabilities by integrating structural and semantic analysis. The Conformer’s hybrid architecture ensures that both local and global code features are modeled effectively, while the LLM’s semantic embeddings provide a deeper contextual understanding, crucial for detecting complex vulnerabilities like SQL injection or cross-site scripting. This tripartite approach—LSTM for sequence modeling, Conformer for structural fusion, and LLM for semantic enrichment—positions our model as a comprehensive tool for block-level vulnerability detection in Python, addressing the shortcomings of prior methods that rely solely on static rules or simpler neural networks.

1.1 Structure of the Dissertation

Chapter 1 provides a short introduction to the work presented in the thesis and describes the motivation and context of the author’s contributions, emphasizing the integration of Long Short-Term Memory [27], Conformer, and Large Language Model techniques for block-level vulnerability detection in Python.

Chapter 2 provides the necessary background for the reader on the history of vulnerability detection modeling and measurement. It also presents the existing standards of the area, including traditional static and dynamic analyses, as well as machine learning approaches like LSTM, Transformer-based Conformers [34], and pre-trained LLMs. These form the starting point for the research work introduced in subsequent chapters.

Chapter 3 discusses recent results in vulnerability detection modeling and its applications, including the author’s contributions across these three modeling paradigms. In this chapter, vulnerability detection modeling is discussed at the system level, presenting a detailed evaluation of currently existing practical approaches for vulnerability detection modeling, covering rule-based tools, LSTM-based sequence modeling, and emerging Conformer and LLM techniques.

Chapter 4 discusses the tool’s implementation, comparing it to similar existing tools and highlighting how Conformers and LLMs enhance precision over LSTM-only approaches. We elaborate on the technical implementation of our model, detailing the preprocessing of Python code from GitHub, the training of the LSTM model for sequential token analysis, the Conformer module for capturing local and global code dependencies, and the LLM integration for semantic feature extraction. This chapter provides a comprehensive view of how these components synergize to detect block-level vulnerabilities, including hyperparameter tuning and architectural choices for each. It includes a detailed evaluation of our model’s performance across LSTM, Conformer, and LLM components, using case studies and empirical data to validate their effectiveness in predicting vulnerability attributes. The chapter concludes with results from testing on real-world Python datasets, comparing the hybrid approach to baseline methods.

In Chapter 5, after showing their advantages and drawbacks, we propose a new hybrid approach that leverages LSTM for sequential analysis, Conformers for structural and global feature extraction, and LLMs for semantic enrichment, eliminating most weaknesses of standalone methods. We introduce our model as a prototype model for Python, detailing its tripartite architecture, followed by empirical validation of the proposed models. This chapter focuses on sequential modeling with LSTM and code embeddings, including data collection from GitHub, processing, embeddings like Word2Vec and FastText, architecture, hyperparameter tuning, and performance evaluation. We also discuss several possible applications of the newly proposed techniques and models, such as bug prediction, development cost estimation, and the impact of design patterns on software maintainability. This chapter also explores the long-term goal of learning the effects of coding practices on software package maintainability, leveraging insights from LSTM, Conformer, and LLM analyses. We round off with pertinent conclusions and suggest future directions for further research, including extending our model’s Conformer and LLM components to other languages and unknown vulnerabilities.

Finally, the appendices contain a summary of the thesis in English, including Thesis Point I and Thesis Point II.

1.2 Summary of the Results

The main results presented in this thesis are related to vulnerability detection modeling and measurement, as well as the application of the newly proposed methods, tools, and techniques in software security. All the novel theoretical results and models were thoroughly validated via empirical case studies and successfully applied in practice. The thesis result statements have been grouped into two major thesis points, where the author’s contribution is clearly shown. The relation between these points and supporting publications is shown in Table 1.1.

I. Sequential Modeling with LSTM and Code Embeddings

The contributions of this thesis point are related to vulnerability detection at the code level using LSTM and various embeddings, discussed in Chapter 3. Embeddings comparison and LSTM baseline. To address the suitability of text representation techniques for vulnerability prediction, we compared word2vec, fastText, and CodeBERT embeddings with an LSTM model on Python code datasets [1]. CodeBERT proved most effective, achieving 93.8% accuracy, while fastText and word2vec showed lower performance. This established a strong baseline for sequential modeling, capturing token dependencies in vulnerabilities like SQL injection. The comparison involved training embeddings on a corpus from popular GitHub repositories, such as numpy, django, and scikit-learn, tokenized into lists of Python tokens with comments removed and indentations adjusted. Hyperparameters like vector dimensionality, minimum token count, and iterations were tuned, with retaining strings outperforming replacement for richer semantic capture. Block-level ML-based detection tool. Building on this, we developed a block-level tool using LSTM with CodeBERT embeddings, achieving average precision of 91.4% and recall of 83.2% across six vulnerabilities on GitHub datasets [3]. Empirical validation on real-world code confirmed LSTM’s utility for fine-grained detection, though limited in structural and semantic depth. The tool processes code snippets from commit diffs, split into overlapping blocks with focus windows of 5 characters and contexts of 200, labeled based on pre- and post-fix versions. Performance metrics, such as F1 scores per vulnerability, highlight its effectiveness, with tuning for neurons, batch size, dropout, and Adam optimizer ensuring balanced overfitting prevention. The author’s contributions. The author collected and mined the datasets from GitHub, implemented the LSTM models, performed the empirical validations, evaluated the results, and implemented prototype tools. The contributions of this thesis point relate to utilizing the models for broader code quality analysis, Interconnection between bugs and refactorings. We explored how refactorings impact bug occurrence, analyzing datasets to reveal that certain refactorings reduce bugs while others may introduce them [2]. This ties into vulnerability detection by showing how code changes affect maintainability and security, with empirical results on open-source systems. The author’s contributions. The author performed the data mining, statistical analyses, evaluations, and drew key conclusions on refactoring effects.

II. Conformer-LLM Integration for Block-Level Vulnerability Detection

The contributions of this thesis point focus on integrating Conformer and LLM for block-level detection, discussed in Chapter 4. Conformer-based hybrid model.

We proposed a Conformer-integrated model to capture local and global dependencies, achieving up to 91.3% F1 score on six Python vulnerability datasets [4]. The Conformer outperformed LSTM baselines by 5-10% in accuracy, validated through ablation studies showing reduced false negatives in structural vulnerabilities like command injection. This model processes graph-based inputs encoded into 128-dimensional vectors, concatenated with 512-dimensional CSEs from UniXCoder, forming a 640-dimensional input per token. Hyperparameters like kernel size, d-model, and d-ffn were tuned, with sinusoidal positional encodings preserving token order. LLM integration. Extending this, we incorporated fine-tuned UniXcoder LLM for semantic enrichment, further improving F1 to 93.0% for SQL injection and 91.0% for XSS, addressing semantic nuances missed by structural models alone. UniXcoder, with 12 Transformer layers and 768 hidden units, is fine-tuned over 30 epochs at a 1e-5 learning rate, freezing lower layers to retain pre-trained knowledge while adapting top layers for vulnerability-specific semantics. The hybrid fusion concatenates outputs into a 1152-dimensional vector, processed through dense ReLU and sigmoid layers for binary classification. Ablation confirmed the tripartite synergy, with full hybrid outperforming LSTM+Conformer by 3-5%. The author's contributions. The author designed the Conformer and LLM modifications, implemented the hybrid architecture, conducted the evaluations on GitHub datasets, and analyzed the results.

Thesis Point	[1]	[2]	[3]	[4]	[5]
I	•	•	•		
II				•	•

Table 1.1: Thesis contributions and supporting publications

[1] Bagheri, A. and Hegedűs, P., 2021, August. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In *Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC)* (pp. 267-281). Cham: Springer International Publishing.

[2] Bagheri, A. and Hegedűs, P., 2022, May. Is Refactoring Always a Good Egg? Exploring the Interconnection Between Bugs and Refactorings. In *Proceedings of the 19th International Conference on Mining Software Repositories* (pp. 117-121).

[3] Amirreza Bagheri and Péter Hegedűs. Towards a Block-Level ML-Based Python Vulnerability Detection Tool. In *the 13th Conference of PhD Students in Computer Science : Volume of Short Papers*, pages 17-20. University of Szeged, 2022

[4] Bagheri, A. and Hegedűs, P., 2024. Towards a Block-Level ML-Based Python Vulnerability Detection Tool. *Acta Cybernetica*, 26(3), pp.323-371.

[5] Bagheri, A. and Hegedűs, P., 2024. Towards a block-level conformer-based python vulnerability detection. *Software*, 3(3), pp.310-327.

2

Background

The National Institute of Standards and Technology (NIST) [1] defines a vulnerability as a flaw in an information system, internal controls, system security protocols, or implementation that a threat source could exploit or activate. In the same vein, vulnerabilities are specific defects or oversights in software that enable attackers to execute malicious actions, including the disclosure or modification of sensitive information, the disruption of systems, or the control of programs, as delineated in standard software security assessment handbooks. The identification of code with security-related defects presents a substantial challenge: how can one accurately characterize susceptible code and differentiate it from clean code? According to research conducted by Shin et al. [69], only a small percentage of defects (approximately 20%) are security-related, which emphasizes the necessity of precise detection methods. This chapter examines the progression of vulnerability detection methods, which have progressed from conventional static and dynamic analyses to sophisticated machine learning methods such as Long Short-Term Memory, Conformers, and Large Language Models. The theoretical foundation of our model, a hybrid framework that is intended to resolve these challenges in Python code at a block level, is built upon these methods.

2.1 Traditional Methods

2.1.1 Static Analysis

Static code analysis involves analyzing a program's form, structure, content, or documentation [1] without executing it, utilizing generalization and abstract rules [1] to identify potential vulnerabilities. This approach can pinpoint the specific cause of a detected issue, offering detailed insights into security risks related to access control, information flow, and improper use of APIs, such as cryptographic libraries [61]. Commonly used tools include FindBugs for Java, Splint and Flawfinder for C, and PyT

(Python taint) for Python, which are adept at detecting a variety of issues including incorrect API usage, performance problems, deadlocks, and security flaws [44] like buffer overflows. Static analysis excels at identifying proper programming techniques and possible flaws early in the development cycle, significantly enhancing code quality by enabling engineers to rectify errors prior to execution.

However, the effectiveness of static analysis tools hinges on the quality of the underlying patterns, abstractions, or rules used to detect issues. As software complexity increases and attackers devise more innovative exploits, defining secure software patterns becomes increasingly challenging. If a specific vulnerability is not addressed in a tool's rule set, it remains undetectable until a new rule is implemented, a process that is labor-intensive and time-consuming when done manually. Automatically generating these patterns is practically unfeasible, and keeping pace with all potential vulnerabilities is difficult due to rapid technological advancements. The accuracy of static analysis is assessed by its false positive rate and false negative rate. Ideally, a tool would be perfectly sound, avoiding false positives, but Rice's theorem demonstrates that significant inquiries about a program's characteristics are undecidable, implying inherent limitations. This suggests that overlooked vulnerabilities (false negatives) are inevitable, preventing static tools from being both comprehensive and precise simultaneously. High false positive rates in many tools lead developers to expend significant effort verifying misclassifications, potentially desensitizing them to alerts and causing genuine flaws to be missed. These computational and practical constraints highlight the need for more advanced approaches like Chess et al., [13].

2.1.2 Dynamic and Hybrid Analysis

Dynamic program analysis involves executing and monitoring a program to identify potential security issues, systematically examining execution traces to uncover vulnerabilities [67]. Unlike static analysis, which operates without running the code, dynamic analysis observes the program's behavior in real-time, potentially catching issues that manifest only under specific conditions. However, this method may not cover all possible inputs and execution paths, leading to the risk of missing several vulnerabilities if test cases are insufficient. Additionally, dynamic techniques often demand significant processing resources, rendering them impractical for analyzing large software systems or extensive collections of codebases. The enormous number of test cases required to achieve comprehensive coverage further constrains their scalability and efficiency [2].

To address some of these limitations, hybrid analysis combines the strengths of dynamic and static methodologies, aiming to reduce false negatives by leveraging complementary insights. For instance, Balzarotti et al. [reference] explore sanitization concerns using a hybrid approach: they first assess static features of a program to identify potential issues, then simulate its behavior with various test input strings to refine the analysis when static results appear imprecise. This synergy enhances detection capabilities by cross-validating findings, such as confirming a statically flagged vulnerability through dynamic execution. Despite these advantages, both dynamic and hybrid analyses remain labor-intensive, requiring significant expertise to design effective test suites and interpret results. The reliance on runtime monitoring and the need for extensive testing environments limit their applicability in automated, large-scale vulnerability detection, prompting the exploration of machine learning alternatives

that minimize human effort while improving scalability and precision [7].

2.2 Machine Learning Methods

Traditional methods for detecting insecure code, such as static and dynamic analyses, depend heavily on attributes predefined by human specialists, necessitating substantial manual effort and expert knowledge to maintain effectiveness. This reliance limits their scalability and adaptability to evolving threats, driving the need for more universal, automated, and unbiased techniques [1]. Data mining and machine learning offer promising solutions by extracting knowledge and patterns from vast datasets, reducing human intervention while enhancing detection capabilities. Data mining involves constructing and managing databases, formatting, filtering, and cleaning data from sources like GitHub [31] repositories, Bugzilla [62], and the CVE database, providing a rich foundation for analysis.

Machine learning enables systems to learn from examples, identifying vulnerability patterns in unprocessed code without explicit programming. Our model leverages three advanced techniques to address distinct aspects of Python vulnerability detection: Long Short-Term Memory networks capture sequential syntactic flows, Conformers fuse local and global structural dependencies, and Large Language Models provide deep semantic understanding.

2.2.1 Deep Learning and Neural Networks

Determining the representation of data and attributes can be difficult in many complex scenarios, particularly when high-level and abstract aspects are required. By connecting complicated ideas to simpler ones, deep learning creates a multi-layered hierarchy that helps computer systems understand complex ideas. To do this, neural networks are created [39]. Similar to the neurons in the biological brain, an artificial neural network is a complex system made up of interconnected components called neurons. Layers of neurons make up the arrangement. The term "Deep Learning" [56] describes learning using neural networks with many layers of neurons. After processing several inputs, each neuron generates a single output value, or activation. The first neuronal layer is termed the input layer, the layers that follow are called buried layers, and the last layer is called the output layer.

By activating or deactivating itself in response to inputs and other neurons' activity patterns, neurons alter the successive layers of neurons until the last layer generates the neural network's output. Finding the proper internal rules and triggers that allow a neural network to exhibit a particular behavior is the aim of learning, or training, the network. Finding the right values for the parameters of mathematical functions that create internal regulations is the task at hand.

Each layer may have a different number of neurons. The last layer might just have one neuron in certain classification scenarios where the output is a single value. A "dense" or fully interconnected layer is one in which every neuron in the dense layer is coupled to every other neuron in the preceding layer. The illustration's fourth layer is dense, while the other layers are all sparse. In a dropout layer, certain neurons are arbitrarily killed by setting their output to zero, which stops them from having an

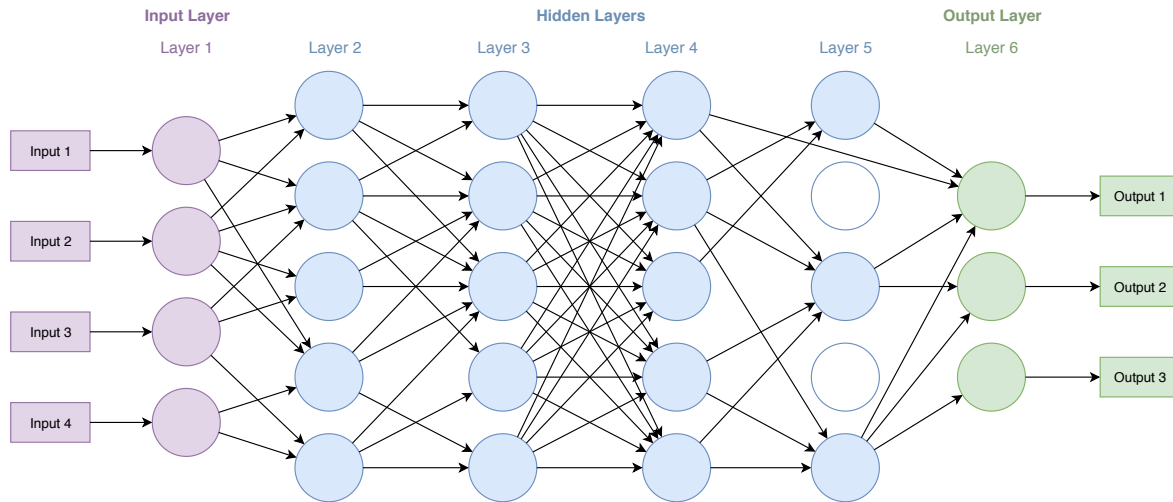


Figure 2.1: Example illustration of a neural network

impact on the layer that follows. This is shown in layer 5 in Figure 2.1. The percentage of neurons to be eliminated is determined by the dropout parameter. Srivastava et al. [72] originally proposed this strategy in 2014 to avoid overfitting, which can happen when a network becomes overly dependent on some nodes. The cost function must take into account the surrounding nodes more by randomly deactivating a section of the nodes.

2.2.2 Recurrent Neural Networks

Neural networks that only accept inputs from the layer above are known as feed-forward neural networks. After successively processing the input data, the networks provide an output. The data never passes through a particular network node more than once. A deterministic output mapping is produced for every input that is sent into the network. As a result, the input alone determines the network's output, which is unaffected by any prior processing. The network cannot take into account the context of the data it is handling. They see every input as completely distinct from the one that comes after it. It is possible to build neural networks so that information flows from later neurons to earlier ones, giving the network a circular shape. These networks use an internal state and are called recurrent neural networks (RNNs), Figure 2.2. Neurons at any layer can change and recover the state, and it can be kept for a long time. It serves as a kind of memory. RNNs can identify dependencies in the data by using a feedback loop to take previous outputs into account during the current operation. RNNs' capabilities are greatly increased by the internal memory, which enables them to recognize patterns in data beyond a single sample. They are able to understand time series data, genetic sequences, musical notes, health data, sensor readings, and the semantic context of written natural text.

For replicating this kind of data, a recurrent neural network with memory is appropriate. RNNs' memory capacities allow them to comprehend similarity in a more sophisticated way. They are able to identify, for example, that two for loops with different counting variables and end conditions are similar structures. There are empirical facts to support the assumptions. Both White et al. [78] and Dam et al. [21] demon-

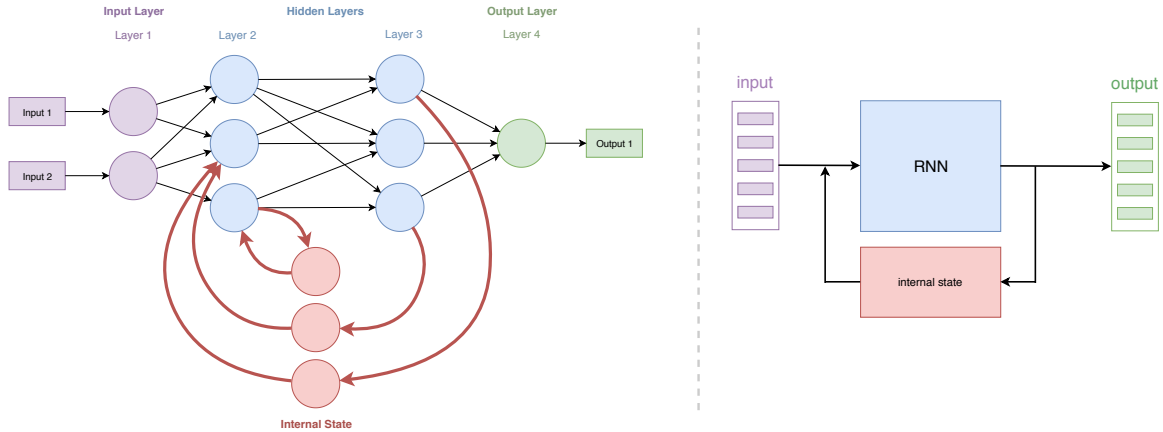


Figure 2.2: Recurrent neural network. Left showing neurons with example connections, right simplified visuals for sequential data

strated the value of recurrent neural networks in source code modeling. RNNs were later successfully used to imitate code clones by White et al. RNNs are not without constraints. They cannot faithfully depict connections across long distances. As the distance between the initial occurrence and later relevance of information rises, recurrent networks that use gradient descent for error minimization become less effective, as demonstrated by Hochreiter [42] and Bengio et al. [8]. The problem of vanishing or bursting gradients is the main cause of this.

2.2.3 Long Short Term Memory Networks

In 1997, Jürgen Schmidhuber and Sepp Hochreiter [43] presented the idea of LSTMs. By using a gradient-based technique that guarantees a steady error flow through internal states, their network architecture solves the vanishing gradient problem and avoids problems like vanishing and inflating gradients. A representation of an LSTM can be found in Figure 2.3. A memory cell is used by LSTMs to retain accumulated context data. The sigmoid neural network layers called 'gates'—an input gate, a forget gate, and an output gate—may be used to change the memory content. Each gate can generate a value between 0 and 1. A value of 1 permits all information to pass through, whereas a value of 0 blocks all information. Every gate is educated and influences the contents of the memory cell, which in turn influences the final output. By choosing which information to keep and which to discard, LSTMs avoid vanishing or exploding gradients.

A sigmoid layer called the forget gate is made to specifically exclude data that is no longer pertinent to the cell state. Using the formula $ft = (Wf[ht1, xt] + bf)$, where $ht1$ is the output of the previous step, xt is the current input, and Wf and bf are weights, the memory cell's state component $Ct1$ outputs a value between 0 and 1 to determine the retention level. This gate is intended to make it easier to eliminate information that isn't relevant in order to stop it from persisting and skewing results in the future. A multiplicative gate, the input gate shields the data in the memory from unnecessary inputs. Finding out which information needs to be updated is the goal. Each of the two halves of the process takes into account weights, the current input, and the output of the previous phase. To determine the new values, one component makes use of a

hyperbolic tangent \tanh gate: The output is limited between -1 and 1 by the formula $Ct = \tanh(Wc[ht1, xt] + bc)$. A sigmoid gate, the second part, controls which values are kept in the memory cell: $(Ht1, xt) + bi$). The product is then added to the memory cell once the numbers have been multiplied.

To select which values are output, the output gate's sigmoid layer computes $ot = (Wo[ht1, xt] + bo)$. After being normalized between -1 and 1 using a \tanh function, these values are then multiplied by the memory cell values. The value $ht = ot \tanh(Ct)$ is used as $ht1$ in the feedback loop for the next phase and is also included in the LSTM output. Although a standard architecture for an LSTM is described in the previous section, there have been a number of modifications created. The forget gate was introduced later on and was not originally included in the original LSTM design. Other variants of LSTMs include gated recurrent units that combine gates, simplifying and changing the LSTM design, and 'peephole connections' that allow gate layers to take into account the current cell memory state.

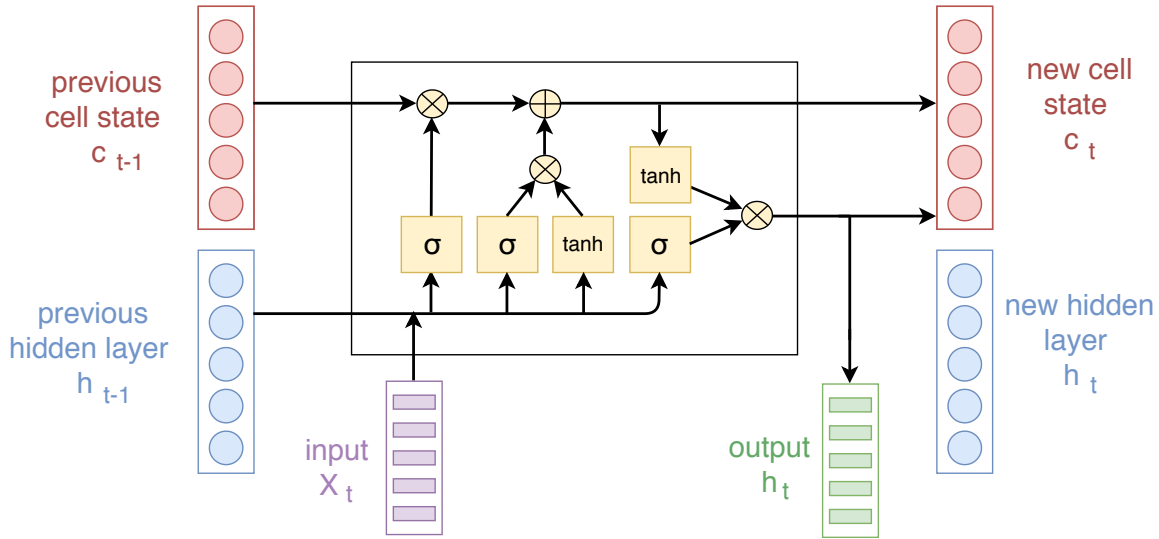


Figure 2.3: LSTM network

2.2.4 Large Language Model

Large Language Models (LLMs), such as BERT [47], CodeBERT [25], and UniXcoder [35], leverage Transformer architectures pre-trained on vast corpora of code and natural language to produce contextual embeddings, marking a significant leap in machine learning for code analysis. Developed by Devlin et al. and refined for code by Feng et al. LLMs like UniXcoder consist of a 12-layer Transformer encoder with 768 hidden units and 12 attention heads, processing up to 768 tokens. Unlike static embeddings like word2vec [15], which assign fixed vectors, LLMs generate dynamic representations using multi-head self-attention:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \quad (2.1)$$

where Q , K , and V are query, key, and value matrices derived from input embeddings, and d_k is the dimension of each head, enabling the model to weigh token

relationships contextually. Pre-training employs masked language modeling [70], masking 15% of tokens for prediction, and next-sentence prediction tasks, building a deep understanding of syntax and semantics across millions of parameters.

In vulnerability detection, LLMs excel at interpreting Python code semantics beyond syntactic patterns. For instance, given a snippet with `user_input = input()` followed by `os.system(user_input)`, an LLM can infer the risk of command injection based on the contextual meaning of `input()` as unsanitized user data, a subtlety LSTM might miss without explicit sequence cues. Similarly, in a database query like `cursor.execute("SELECT * FROM users WHERE id = " + user_id)`, the LLM recognizes the lack of parameterization as a potential SQL injection vulnerability, leveraging its pre-trained knowledge of secure coding practices. Our model fine-tunes UniX-coder on GitHub-derived Python vulnerability data, enhancing its ability to detect context-dependent flaws like path disclosure or API misuse, where semantic intent is key.

Despite their power, LLMs face challenges: their large parameter size demands significant computational resources, exceeding even the Conformer’s requirements, and pre-training on general corpora may not fully align with Python-specific vulnerabilities without extensive fine-tuning. Additionally, their focus on token-level semantics might overlook structural dependencies better captured by Conformers, necessitating their integration in our model to achieve comprehensive block-level detection.

2.3 Representing Code

To provide important structural insights, we use data flow graphs, control flow graphs, and abstract syntax trees as inputs to our computational model. Structured examination of inserted code snippets requires graph-based frameworks. When it comes to code analysis, graphs offer many benefits. They make it simpler to understand data relationships and control flow patterns by condensing the essential structural components of code. Finding complex relationships across the software ecosystem is made possible by this all-encompassing viewpoint. By eliminating superfluous parts of the code and simplifying the key parts, graphs offer abstraction. The analytical difficulty is greatly decreased by this simplification.

Because they offer a structured input that facilitates predictive modeling, graph-based formulations are naturally compatible with machine learning paradigms and other computational techniques. Several widely used research methodologies are described in this section and will be cited later. This is by no means an exhaustive list of all possible methods.

2.3.1 Abstract Syntax Trees

Abstract syntax trees (ASTs) offer a hierarchical depiction of code or code modifications, which can be utilized in various formats by excluding varying numbers of tokens to streamline the tree. Yamaguchi et al. [80] present many approaches: restricting nodes in the tree to solely API nodes and function names, utilizing subtrees of depth n with API nodes and placeholders, or employing subtrees that consist of API nodes, placeholders, and syntax nodes. Often, significant sections of the code are not

included in the Abstract Syntax Tree. Many scholars have put significant effort into constructing complex Abstract Syntax Trees from their code and manipulating them. It has been suggested that mining patterns from plain text code is too difficult, hence the use of AST notation is essential. It should be noted that there are counterexamples demonstrating that working with plain text can also be a highly effective method.

2.3.2 Control Flow Graph

A basic block, which is a sequential segment of code devoid of jumps or jump targets, is represented by each node in a control flow graph. A block's start and finish are indicated by jump targets and jumps, respectively. The control flow transitions are represented by directed edges. The entry block, where control enters the flow graph, and the exit block, where all control flow exits, are the two designated blocks found in the majority of presentations. [82] Because of the way a CFG is created, each edge A-B has a unique *attribute*. Either the *indegree* of B or the *outdegree* of A is larger than 1, or both. The program's whole flow graph, in which each node represents a single instruction, can be used to create the Control Flow Graph. Every edge that invalidates the predicate is then subjected to an edge contraction, which is the contraction of edges where the destination has a single entry and the source has a single exit. With the exception of being a tool for visualizing the CFG's development, the contraction-based approach has no practical significance. Directly searching the program for basic blocks can produce the CFG more quickly.

2.3.3 Data Flow Graph

A data flow graph is a graphical representation of computer programs that illustrates potential for simultaneous execution of program components. Nodes in a data flow graph, referred to as actors, symbolize operations and predicates that are applied to data objects. Arcs in the graph indicate channels over which data objects pass from a producing actor to a consuming actor. Both the control and data parts of a program are depicted in a unified model. An actor is considered enabled when data items are present at its input ports and specific conditions are met. Data flow models reveal the parallelism within a computation by allowing actors to be enabled to fire simultaneously or sequentially in any order, which can be utilized in implementing the model. Various iterations of data flow graphs [23] have been examined in research, although they possess notable similarities. A DFG is a directed graph where an arc serves as a pathway for data transfer from a generating node to a consuming node. A node in a DFG operates by receiving data items from its inputs, executing computations, and then sending the resulting data items to its outputs, similar to how a Petri net fires. A node's action is initiated by the existence of input data.

Research on data flow graphs has mostly concentrated on three distinct formal models: static data flow, dynamic data flow, and synchronous data flow. The different models vary in terms of the number of data items allowed on an arc, the allowance for actors to have internal state, and other specific specifics.

2.3.4 Code as Natural Text

Code can be seen as a sequence of words or tokens that closely resembles natural language. Recently, there has been a growing trend in study utilizing Natural Language Processing methods on software code, considering it as natural language text. Code features can be automatically extracted, allowing for a variety of applications [22].

The 'natural hypothesis' of code posits that coding is a type of communication, and extensive code collections exhibit patterns akin to those found in natural language. The initial evidence supporting this concept was presented by Hindle et al. [41] Source code exhibits similarities with natural language text in terms of repetitive structures, common patterns, local repetitions, and long-term dependencies. Code is often created by humans who have a tendency to favor conventional, known, and typical patterns and structures, leading to their emergence. Machine learning models effective in natural language modeling can also be used to code due to their power in identifying and generalizing trends while managing noise. Insecure code frequently harbors multiple vulnerabilities that have a common defective pattern, such as the absence of a check before a function call. Identifying the similarities of vulnerabilities can readily lead to their detection. Another crucial aspect of code is its 'localness,' [75] which refers to the frequent occurrence of specific patterns within a relatively narrow range.

Code is distinct from normal language in that it is 'semantically brittle,' [5] meaning that even little modifications to the code can significantly modify its usefulness. The data also includes intricate structural details, such as nested loops and similar constructions. NLP-inspired models have been effectively applied to software code, despite the differences.

2.4 Embedding Code in a Numerical Vector

To train a neural network on the data, the samples must be converted into a numerical representation. Only a brief summary is provided above. Researchers primarily use handmade approaches to convert the tree structure of Abstract Syntax Trees, Control Flow Graphs, and Data Flow Graphs into numerical vectors.

2.4.1 Word2Vec Embedding

The word2vec embedding was created in 2013 [15]. It provides a distinct vector representation for each token, considering semantics by assigning comparable tokens similar representations. Word2vec is a two-layer neural network that requires training on a specific corpus of data to understand the semantics. Word2vec utilizes the co-occurrence and relationship of words to assign vectors that have high cosine similarity to tokens that are semantically comparable. If two words are completely unrelated, their vectors would be perpendicular, while full similarity is represented by a 0-degree angle. Code tokens are intended to have better similarity between vectors for true and false compared to return and while.

All vectors within a specific embedding share identical dimensions or cell count. The encoding utilizes cells to encode specific semantic characteristics. In a word2vec model

working with the English language, the 3rd cell may signify that a word represents an item, showing high values in the embeddings of words like apple or book, but not in words like such or leave. The cells' representation is learned automatically by the word2vec model, and their actual meaning may only be inferred later. It is usually not feasible to reverse engineer the function of each cell in a given word2vec model for most applications. Additionally, there are other similar embedding models that perform equally well on the same dataset.

Figure 2.4 depicts a simple depiction of a word2vec model. Python code tokens such as "true" or "count" are converted into a vector representation with a dimensionality of eleven, shown as eleven colored cells for each token. Numerical values in a vector are depicted using colors to visually emphasize the distinctions and similarities between them. True and false have comparable vector representations in the first to fourth cell in this example. Additionally, it is possible to speculate that cells 8, 9, and 11 may encode certain aspects of logical operations, given their resemblance in values to the token representation of "if" and "and". It is not possible to establish a precise and unambiguous definition for every cell.

A simplified two-dimensional vector space is depicted on the far right of the picture, having the encoding for words expressed symbolically as two-dimensional vectors. Tokens with linked semantics have high cosine similarity, indicated by a narrow angle between them. In a real word2vec implementation, the vector space often exceeds 100 dimensions.

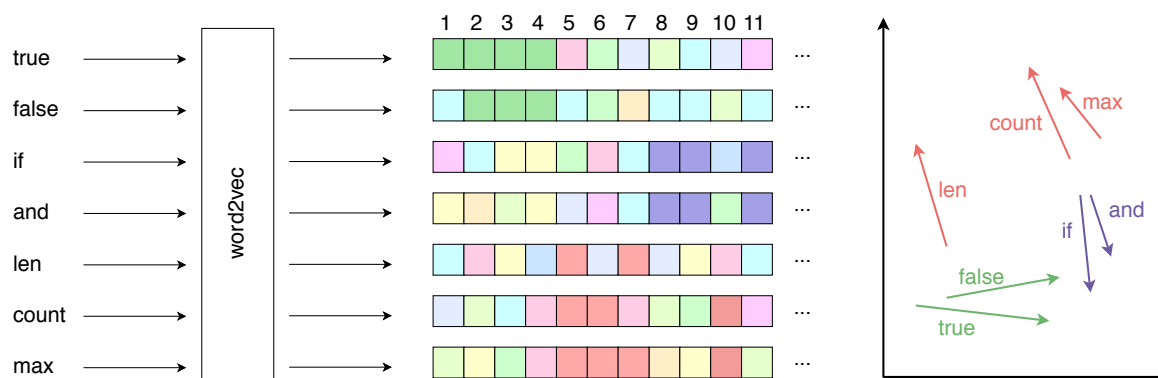


Figure 2.4: The word2vec embedding

2.4.2 FastText

FastText [45] is a compact package created to facilitate the development of scalable solutions for text representation and classification. The program is compatible with standard, generic hardware and may be used on smartphones and small PCs by utilizing a feature that minimizes the memory usage of fastText models. FastText is designed to be user-friendly for developers, domain experts, and students. The main goals of this technology are word representation learning and text categorization. Without the need for specialized gear, it was designed to facilitate quick model iteration and refinement. A multicore CPU may be used to train fastText models on more than a billion words in a matter of minutes.

The dataset contains pre-trained models that have been trained on Wikipedia and in more than 157 different languages. fastText can be utilized through a command line

interface, integrated with a Python program, or employed as a library for a wide range of applications, spanning from experimentation and prototyping to production.

2.4.3 CodeBERT

A variety of natural language programming tasks, such as code search and documentation creation, are made easier by CodeBERT [25]’s acquisition of multidimensional representations. In order to identify suitable alternatives generated by samplers, CodeBERT was created using a Transformer-based neural architecture and trained using a hybrid objective function that includes the pre-training job of replacement token recognition. We can use unimodal data as well as bimodal data made up of NL-PL pairs. While the unimodal data improves the generators, the NL-PL pairs supply input tokens for model training. By altering model parameters, we assess CodeBERT’s performance in two applications that translate natural language to programming languages. The results show that CodeBERT is more effective in both code documentation production and natural language code search activities. In order to explore the type of information gathered in CodeBERT, we also create a dataset for NL-PL probing. We assess this in a zero-shot scenario, in which the pre-trained model’s parameters remain unchanged.

2.5 Self-Attention Based Methods

Self-attention mechanisms enhance sequence modeling by enabling parallel processing and capturing dependencies across entire sequences, overcoming limitations of recurrent architectures like LSTM. In vulnerability detection, these methods excel at analyzing code relationships beyond local contexts, forming a critical component of our model’s hybrid approach alongside LSTM and LLMs. This section explores two key self-attention-based techniques: Transformers, which establish the foundation, and Conformers, which refine it for structural and global feature extraction.

2.5.1 Transformer

Introduced by Vaswani et al. [76], the Transformer architecture replaces recurrent processing with a mechanism called self-attention, allowing it to process entire sequences simultaneously rather than step-by-step like LSTM. It uses multiple layers, each containing a self-attention module that weighs the importance of every token relative to all others, followed by a feed-forward neural network. To keep track of token order without recurrence, Transformers add positional information to the input, typically through a fixed pattern rather than learning it from the sequence itself. This parallel approach makes Transformers highly efficient at capturing long-range relationships in data.

In Python vulnerability detection, Transformers analyze code tokens to identify connections across distant parts of a program. This capability is ideal for spotting risks that span functions or blocks, such as an XSS vulnerability where unescaped data ends up in HTML output far from its source. Unlike LSTM, which processes tokens

sequentially and may miss such distant dependencies, Transformers handle these relationships naturally. However, their efficiency comes at a cost: they require significant computational resources, especially for long code snippets, and their focus on broad context can sometimes overlook finer local patterns, like subtle syntactic errors within a single block. In our model, Transformers serve as a foundational step, improved upon by Conformers for more precise structural analysis.

2.5.2 Conformer

The Conformer, introduced by Gulati et al. [33], represents an evolution of the Transformer architecture by integrating convolutional neural networks with self-attention mechanisms, designed to overcome limitations in modeling both local and global dependencies efficiently. Unlike traditional Transformers, which rely solely on self-attention and struggle with long sequences due to their quadratic computational complexity, the Conformer incorporates a convolutional feed-forward module alongside multi-head self-attention. This hybrid design includes four key components within each block: a feed-forward module, a multi-head self-attention module, a convolutional module, and a second feed-forward module, all stabilized by layer normalization. The architecture employs sinusoidal positional encodings to preserve token order, calculated as:

$$\text{pos}_i = \begin{cases} \sin\left(\frac{\text{pos}}{10000^{2i/d}}\right), & \text{if } i \text{ is even} \\ \cos\left(\frac{\text{pos}}{10000^{2i/d}}\right), & \text{if } i \text{ is odd} \end{cases} \quad (2.2)$$

where pos is the token position, i is the dimension index, and d is the embedding dimension, ensuring the model captures positional context effectively.

In vulnerability detection, the Conformer processes Python code tokenized into sequences, leveraging its convolutional module to extract local syntactic patterns while the self-attention mechanism models global dependencies, like variable usage across distant function calls. This dual capability is critical for block-level analysis, where vulnerabilities often arise from interactions between local syntax and broader program logic. For instance, a Conformer can identify a command injection risk by linking a local string concatenation to a shell-executing function elsewhere in the code, a task where LSTM's sequential focus might falter. The attention mechanism uses a modified scaled dot-product formula to reduce noise:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{1 + \sqrt{d_k}}\right)V \quad (2.3)$$

where Q , K , and V are query, key, and value matrices, and d_k is the embedding dimension, enhancing stability over the standard Transformer approach.

Despite its advantages, the Conformer's complexity increases computational demands compared to LSTM, requiring significant resources for training and inference. Additionally, while it excels at structural analysis, it may lack the semantic depth of LLMs, necessitating a complementary approach in our model to fully address Python code vulnerabilities.

2.6 Mining Software Repositories

The effectiveness of machine learning methods is heavily influenced by the quality and attributes of the training data [59]. Thus, obtaining an extensive array of top-notch examples is the initial and essential stage in all machine learning processes using source code. Version control and source code management solutions are commonly used to track problems and code changes in software systems [85]. They facilitate code sharing among developers and provide public access in the context of open source initiatives.

The increase in the scale, quantity, and appeal of publicly available projects on hosting platforms has enabled the accumulation of extensive code repositories, facilitating the use of data-driven methods to identify vulnerabilities [63]. Unlike studies that concentrate on a small number of projects for model training and validation, it is possible to utilize hundreds or even thousands of repositories to derive broader findings. Moreover, the repositories typically consist of real-world projects of genuine apps, rendering them more pertinent than vulnerability databases including artificially created instances. Consequently, a significant amount of substandard code will also be found. In addition to analyzing the code itself, software repositories can also be examined to get information on dependencies, commit messages, change trends, and other metrics [50].

2.6.1 Approaches for Mining Software Repositories

Kagdi et al. [46] categorized techniques into mining by annotations, via heuristics, via differencing, and via data mining. The initial method concentrates on commit messages and metadata to assess factors such as frequency of class changes, correlation between class changes, occurrence rate of changes in specific subsystems, and more.

The second approach emphasizes heuristics related to developers or code structure. Mining via differencing involves analyzing the discrepancies between two versions of source code, such as identifying the number of inserted or deleted functions or function calls, altered conditions, or changes in abstract graph representation. Data mining identifies cross-connections between changes and associations. Zimmer et al. [87] examined combinations of functions frequently altered simultaneously to provide recommendations, aiming to prevent developers from making mistakes caused by incomplete modifications and anticipate probable changes. Several methods consider high-level information such as classes, functions, and metadata, whereas differentiation and Data Mining might potentially be utilized at a very detailed level, even at the level of individual code statements.

2.6.2 Tools for Mining Software Repositories

Mining software repositories is an activity that is not conducive to manual labor, thus necessitating the employment of tools for assistance. The primary functions of those tools are data extraction, data filtering, pattern identification, and prediction. Chaturverdi et al. [10] offer a summary of available tools and datasets utilized in mining software repositories. They observe that data retrieval and preprocessing are often the most time-consuming aspects of the entire process.

2.6.3 Mining Github

GitHub offers hosting services that are built on the Git version control and source code management framework. In the summer of 2024, Github [31] had 420 million repositories, over half of which were public open source projects, and over 100 million developers. Currently, a search for public Python repositories yields about 5 million results. Allamanis et al. [5] regard to large amounts of code data as 'Big Code', which can be utilized to acquire insights about features present in vulnerable code.

Information sourced from Github is commonly utilized in contemporary research to educate machine learning models in identifying defects and susceptibilities, as referenced in [85], [63], and [50]. The baseline rate of security-relevant commits is minimal. Therefore, efforts to collect a substantial dataset of vulnerable and patched code should adapt their mining strategy by utilizing security-related search terms [85].

2.7 Python Vulnerabilities

Common vulnerabilities in source code may be present in multiple languages and application domains, while some are exclusive to certain cases. Buffer overflows are common in the C family of languages, while cross-site request forgery is relevant only in the context of online applications. The our model method outlined in this study targets Python programs. This section will introduce and describe common vulnerabilities that will be discussed later. All provided examples are very basic and are only used to demonstrate the main concept. In reality, exploits are usually more intricate.

2.7.1 SQL Injection

As per the OWASP foundation [26], SQL injections are considered one of the most prevalent and severe security issues impacting web applications. The Common Weakness Enumeration defines a SQL injection as the following: The software generates a SQL command using input from another component, but fails to properly neutralize specific elements that could alter the SQL command when it is passed to the next component [17]. Failure to sanitize user-controllable input may result in the input being treated as a SQL statement and executed, rather than as user data. This vulnerability can be used to manipulate searches, such as gaining unauthorized access to files or inserting extra statements to modify or delete databases. Any database-driven website is at risk of being targeted by an exploit if proper sanitization is not implemented. The next excerpt demonstrates this vulnerability.

```
(...)
cur = db.cursor()
name = raw_input("Enter name: ")
cur.execute("SELECT * FROM users WHERE username = " + name +
            """;")
(...)
```


When a user submits a valid request, such as "Tom" everything proceeds smoothly without any adverse consequences. When a user inputs a string with SQL code, such as Tom; DROP TABLE users;, the semicolon is seen as the query's conclusion, causing everything after it to be treated as a separate command, resulting in the deletion of the whole database table. Modifying the code as follows:

```
(...)
cur = db.cursor()
name = raw_input("Enter name: ")
cur.execute("SELECT * FROM users WHERE username = %s;", (name,
))
(...)
```

The parameter following the comma is escaped and then inserted in place of the placeholder s. SQL code tokens have been eliminated, making the request safe for execution. This is just one solution to address the issue, but it highlights the overall necessity of filtering and cleaning user input.

2.7.2 Command Injection

Referring once more to the Common Weakness Enumeration: The software takes input from an upstream component to build a command, but fails to properly neutralize specific parts that could alter the command when it is passed to a downstream component [20]. This scenario involves executing untrusted data on a system to target a command executed by the server shell, rather than a SQL database. This can empower an attacker to read, modify, or delete files that they are not supposed to access. Below is a basic example of a command injection vulnerability in Python 3:

```
import subprocess
filename = input("Please provide the path for the file: ")
command = "cat {path}".format(path=filename)
subprocess.call(command, shell=True)
```

When a user inputs file.txt, the file is shown using the cat command. However, by including a semicolon, the user can add further commands that will be run without any confirmation. Entering file.txt; ls would display the current directory in the shell, while file.txt; rm -rf / would result in significant harm. To prevent this vulnerability, avoid using subprocess with shell=True and sanitize or filter the input.

2.7.3 Remote Code Execution

Remote Code Execution is a scenario where programming code is run on the target system, as opposed to command injection when an OS system command is performed. At times, it can refer to a hacking objective rather than a weakness, when taking advantage of a vulnerability allows the attacker to run any instructions on a system.

2.7.4 Cross-Site Scripting

Cross-site scripting (XSS) is a critical vulnerability in web applications. It is a common occurrence among the OWASP top ten vulnerabilities . Unsanitized data is the primary cause of cross-site scripting vulnerabilities. User-inserted custom code is put to a website or URL and distributed to other users, who will subsequently receive and execute the code in their browser.

Cross-site Scripting is defined by the CWE as: The software fails to properly neutralize user-controllable input before it is included in output that is used as a web page served to other users. An example of a classic vulnerability is stored cross-site scripting, which can occur in a guest book that allows unauthorized input. Visitors can input plain text or Javascript code, which will be saved on the site and shared with other visitors for execution. Various modifications are available, such as utilizing diverse input options and generating executable code using Flash and other programming languages. Another instance involves an email with a hyperlink to a website that harbors harmful Javascript code in the URL, which will run if the link is clicked. User-generated material must be sanitized to prevent XSS attacks, utilizing functions like `html-escape` and others. An illustration from the writings of Micheelsen et al. [52] demonstrates a susceptible portion of Python programming utilizing the web framework Flask. The user-provided input `param` is incorporated into the result page using the `html.replace` method.

```
@app.route("/ XSS_param",methods =["GET"])
def XSS1():
    param = request.args.get("param","not set")
    html = open("templates / XSS_param.html").read()
    resp = make_response(html.replace("{}{param}}",param ))
    return resp
```

2.7.5 Cross-Site Request Forgery

Cross-site request forgery is defined by the CWE as: The web application fails to adequately authenticate if a properly structured, legitimate, and coherent request was intentionally sent by the user [19]. and elaborates: If a webserver lacks a way to validate the purposeful sending of a request from a client, it could be vulnerable to attackers tricking clients into making inadvertent requests that are accepted as authentic. This can be accomplished by a URL, picture loading, XMLHttpRequest, etc., potentially leading to data exposure or unintentional code execution. The user may have initiated this request by willingly clicking on an HTML form, or a bad individual could have deceived the user into hitting a link with identical parameters that initiates the same POST request. The password is altered, potentially resulting in the user being locked out of their account while the attacker gains access. To prevent this type of attack, utilize unique and confidential tokens shared between the client and server for each request. An example is provided to demonstrate the utilization of a token in order to mitigate XSRF attacks.

```
from oauth2client import xsrfutil
(...)
def CheckToken(self, *args, **kwargs):
    user = users.get_current_user()
    token = str(self.request.get("xsrf_token"))
    if not user or not xsrfutil.validate_token(_GetSecretKey(),
    token, user.user_id()):
    self.abort(403)
```

2.7.6 Directory Traversal / Path Disclosure

A path traversal or directory traversal vulnerability arises when a person may manipulate input to reveal file system routes that were not intended to be accessed. The software constructs a pathname using external input to identify a file or directory under a restricted parent directory. However, it fails to neutralize special elements in the pathname, which can lead to the pathname resolving to a location outside the restricted directory [18]. An example of this vulnerability is a website that shows a file based on a path supplied in a URL parameter. By altering this option to include `'../..'`, the attacker can traverse the file system and potentially reveal files that were not intended to be viewed. The code snippet below demonstrates a vulnerability, assuming that `"filepath"` is a user-provided variable.

```
import os.path
(...)
prefix = "/home/user/files/"
full_path = os.path.join(prefix, filepath)
read(full_path, "rb")
```


3

Related Work

The following section discusses prior research on identifying vulnerabilities and includes an attempt at classification, considering several criteria for comparison. The pros and cons of the prior approaches are outlined, with our model being classified among them.

3.1 Traditional Approaches

Traditional approaches involve algorithmic matching and manual verification, without using artificial intelligence. Initially, in the early phases of traditional vulnerability detection, specialists created high-quality rule bases manually. To reduce the financial strain of creating these rule bases, researchers have implemented semi-automated methods such as taint analysis, symbolic execution, fuzzing, and code similarity-based vulnerability matching. These methods successfully reach their goals but frequently face challenges due to incomplete automation and increased labor expenses.

3.1.1 Vulnerability Prediction Based on Software Metrics

What features should be utilized for forecasting the vulnerability of code? Traditionally, the most frequently utilized features were located externally to the source code, in the shape of software and developer metrics. The factors mentioned are size of the code (LOC), cyclomatic complexity, code churn, developer activity, coupling, number of dependents, and legacy metrics [53]. These metrics are commonly utilized as characteristics for constructing fault prediction models [37] and are crucial in the realm of software quality and reliability assurance. Nagappan et al. [54] utilize organizational measures to forecast software failures. It is conceivable that those measurements could be utilized for vulnerability prediction, although there are several issues with that approach.

Two pieces of code may have identical metrics, such as complexity, yet exhibit completely distinct behaviors, resulting in varying vulnerability levels. Moreover, they also have a tendency to not generalize effectively across different software projects. The main critique is that these metrics fail to encapsulate the semantics of the code and

do not consider the actual source code, program behavior, or data flow. The method effectively assumes a predetermined outcome that specific meta traits will be connected to security vulnerabilities, which may not always be the case [44]. For instance, several vulnerabilities might potentially occur in basic applications. Often, the simple solution to an algorithmic problem may lack the necessary safeguards to prevent exploits. This is why software developers facing time constraints or lacking experience in security considerations encounter issues. Code complexity is not a reliable indicator of security vulnerabilities, and the same may be said for other measures. Nevertheless, it is important to recognize that software metrics can provide valuable insights. The statement is exemplified by works that employ machine learning techniques using code metrics to forecast the presence of security vulnerabilities in software.

Shin et al. [69] utilized nine complexity criteria to forecast vulnerabilities in Javascript projects, resulting in a low rate of false positives but a somewhat high rate of false negatives. The authors utilized code complexity, code churn, and developer data in a subsequent study to forecast vulnerabilities. They attained an 80% recall rate and 25% false positives using linear discriminant analysis and Bayesian networks [68].

Chowdhury et al. [14] attempt to forecast software vulnerabilities by utilizing complexity, coupling, and cohesion metrics that have previously been used for fault detection. A study was conducted on Mozilla Firefox versions using decision trees, random forest, logistic regression, and naive Bayes models to identify vulnerabilities. The models achieved approximately 70% precision and recall.

Zimmerman et al. [86] expanded the list by examining code churn, code complexity, code coverage, organizational measures, and actual dependencies. A weak yet statistically significant link was discovered between the analyzed measures. Logistic regression was employed to forecast vulnerabilities based on these indicators, with a specific focus on the proprietary code of Windows Vista. The measures accurately predicted vulnerabilities with a median precision of 60% but had a comparatively low recall of 40%. Neuhaus et al. examined import statements in the Mozilla project and found that using support vector machines to forecast vulnerabilities based on import statements resulted in an average precision of 70% and recall of 40%.

Yu et al. [83] consider many features, including software metrics like number of subclasses and methods in a file, crash features, and code tokens with associated tf-idf scores. They utilize a combination of several perspectives in their approach. They anticipate weaknesses at the file level and effectively reduce the amount of code that needs to be reviewed by human experts to identify vulnerabilities. Other researchers have successfully made predictions only based on commit messages. Zhou et al. [85] utilize a K-fold stacking technique to examine commit messages for predicting the presence of vulnerabilities, achieving significant results. Russel et al. [63] discovered that both humans and Machine Learning systems were ineffective in predicting build failures or defects just based on commit messages. The our model technique does not consider external code metrics but instead learns features directly from the source code.

3.1.2 Anomaly Detection Methods for Finding Vulnerabilities

Anomaly detection involves defining typical and anticipated behavior and identifying deviations from it. Noncompliant code is assumed to frequently be the root cause of defects. Data mining techniques have been utilized to analyze source code and discover typical coding patterns. Li et al. [49] created a tool named PR-Miner that can

identify code patterns in various programming languages and has been demonstrated to be highly effective. Their method focuses on correlating programming patterns that are commonly used together, regardless of the programming language. Bugs identified by their tool have been verified in Linux, PostgreSQL, and Apache. One major issue is that vulnerabilities that represent general patterns in the code are often missed, leading to the failure to identify prevalent weaknesses [80]. Rare programming patterns or API usages may be incorrectly identified as false positives due to their infrequent occurrence. Many anomaly detection methods exhibit significant percentages of false positives [28]. Anomaly detection in code is not a straightforward method for identifying security vulnerabilities, as distinguishing between frequent code pattern violations that affect security and those that do not is challenging. This work’s method to anomaly identification deviates from the norm by using explicit labels to train a model on susceptible and secure code, thus sidestepping the assumption that ‘normal’ is synonymous with ‘right’.

3.1.3 Vulnerable Code Pattern and Similarity Analysis

When aiming to identify vulnerabilities, it is more practical to focus on recognizing the characteristics of vulnerable code rather than delving into theoretical metrics or correct code definitions. Two methodologies can be used to answer that question: susceptible code pattern analysis and similarity analysis. Similarity analysis performs the task implied by its name. The objective is to identify code snippets that are most similar to a susceptible code snippet, as they are likely to also have the same vulnerability. This method is most effective for identical or almost identical code clones with extremely similar intrinsic structures, a common occurrence, particularly through code sharing in the open-source community. Susceptible code pattern analysis involves analyzing susceptible code segments using data mining and machine learning techniques to identify their characteristic features. The features indicate patterns that can be utilized on fresh code segments to detect vulnerabilities. Many studies in this field collect a substantial dataset, analyze it to derive feature vectors, then apply machine-learning methods, as outlined by Ghaffarian et al. [28].

Both methods are usually used on source code without running it, as a static analysis, while some academics also integrate their method with a dynamic analysis. Unlike traditional static analysis, automatic or semiautomatic methods are used to generate features, removing the need on subjective human experts. An unbiased model can be constructed by directly learning from a dataset of code to understand what susceptible code consists of. Often, these methods depend on a broad level of detail, categorizing entire programs [32], files [68], components [55], or functions [81], which hinders the ability to identify the precise position of a vulnerability. Li et al. [48] and Russell et al. [63] employ a more detailed representation of the code. Moreover, the methods vary in several aspects, including the language utilized, the data source, dataset size, label creation process, analysis granularity, machine learning model employed, types of vulnerabilities examined, and the model’s applicability to cross-project predictions versus being limited to the project it was trained on.

Initially, we shall outline fundamental methods utilizing several machine learning methodologies. Subsequently, methods that utilize deep learning are further analyzed. Morrison et al. [53] investigated security weaknesses in Windows 7 and Windows 8 using different machine learning methods such as logistic regression, naive Bayes, support

vector machines, and random forest classifiers.

The study yielded unsatisfactory outcomes, as it achieved notably low precision and recall values. Pang et al. [59] utilize labels from an online database and employ a combination of feature selection and n-gram analysis to categorize entire Java classes as either vulnerable or not vulnerable. Using a small dataset of four Java Android applications, they utilize a basic n-gram model along with feature selection methods to merge related characteristics and decrease the consideration of irrelevant features. They selected support vector machines as the learning technique and achieved approximately 92% accuracy, 96% precision, and 87% recall inside the same project. In cross-project prediction, when training is done on one project and vulnerable files are classified in another, the results were around 65%. Shar et al. [66] utilize machine learning to decrease the occurrence of false positives while identifying XSS and SQLI vulnerabilities in PHP code. They manually select some code attributes and then use a multi-layer perceptron to enhance static analysis tools. When compared to a static analysis tool, they found fewer vulnerabilities but also had lower false positive rates, resulting in an overall satisfactory outcome. In their subsequent research [67], they employed a hybrid methodology incorporating dynamic analysis, which led to significant enhancements in their earlier findings, as demonstrated on six large-scale PHP projects. They also explore unsupervised predictors, which are less precise but remain a fascinating field of study.

Hovsepyan et al. [44] examine unprocessed source code as textual data. They chose an Android email client written in Java as an example and primarily concentrated on evaluating the source code as if it were a natural language, processing files in their entirety. After removing comments, they convert files into feature vectors consisting of Java tokens together with their corresponding frequencies in the file, following a bag-of-words methodology. The feature vectors are categorized into a binary system as either vulnerable or clean. The classifier, a support vector machine, is trained to estimate the vulnerability of a file. This classifier has attained an accuracy of 87%, with precision of 85% and recall of 88%. Their achievement demonstrates that valuable insights can be obtained by examining source code directly as plain text, without the need for complex code representation models. Their work is constrained by being limited to a single software source. In a subsequent project, they applied decision trees, k-nearest-neighbor, naive Bayes, random forest, and support vector machines for a comparable objective [64].

3.2 Alternative Approaches

This section explores modern, machine learning-based vulnerability detection methods that represent alternatives to the traditional approaches. These methods leverage automation and data-driven techniques to improve upon the scalability and objectivity limitations of manual or rule-based systems.

3.2.1 Machine Learning-Based Approaches

Machine learning methods provide automated categorization as a central function, improving vulnerability identification and decreasing the need for manual work. Notable methods in this category comprise Logistic Regression, Multi-Layer Perceptron, Support Vector Machines, and Random Forest. Al-Yaseen et al. [4] introduced a

Paper	Graph-Based	Deep Learning	Large LMs	Manual Meth.	IoT-Specific
Vuldeepecker	✓	-	-	-	-
FUNDED	-	✓	-	-	-
LineVD	-	✓	-	-	-
Code2Vec	-	✓	-	-	-
CodeBERT	-	✓	-	-	-
GraphCodeBERT	-	✓	-	-	-
CuBERT	-	✓	-	-	-
Py150	-	✓	-	-	-
Llama, CodeX	-	-	✓	-	-
GPT-4	-	-	✓	-	-
Cheshkov et al	-	-	✓	-	-
ChatGPT	-	-	✓	-	-
Flawfinder, Vrust	-	-	-	✓	-
Zolanvari et al.	-	-	-	-	✓
Presented Model	✓	✓	✓	✓	-

Table 3.1: Different Approaches

multi-level hybrid intrusion detection approach that utilizes Support Vector Machines and extreme learning machines. They also enhanced the training datasets by utilizing a customized k-means method. Ghaffarian et al. [28] performed a comparative analysis of machine learning techniques for vulnerability identification, emphasizing the higher effectiveness of the Random Forest approach. Lomio et al. [51] performed an empirical investigation to compare the efficacy of several machine learning methods for vulnerability detection and found that the existing metrics may be insufficient. Furthermore, this research acknowledges that ensemble-based classifiers may achieve superior performance. Zolanvari and colleagues [88] evaluated the appropriateness of using machine learning for detecting vulnerabilities in Internet of Things (IoT) environments. Although effective, these strategies may struggle when dealing with complex situations.

3.2.2 Deep Learning for Vulnerability Prediction

Deep learning techniques utilize complex neural networks to enhance the effectiveness of vulnerability detection models in handling intricate problems. Li et al. presented Vuldeepecker [48], a system that uses flat language sequences from source code to train neural networks. Yet, this method compromises the subtle meanings within the code. Therefore, alternative research has investigated the utilization of graph or tree structures, such as the Abstract Syntax Tree or Control Flow Graph, for training. Allamanis et al. [5] proposed methods for converting source code into graphs and for enhancing gated graph neural networks. Wang and colleagues introduced FUNDED, a graph-based approach designed for detecting vulnerabilities. Steenhoek et al. [73] conducted an empirical investigation to analyze the connection between predictions from various SOTP models and the impact of dataset size on the performance of these

models using popular datasets.

Hin et al. [40] utilized LineVD based on graph neural networks to investigate the use of graph neural networks for vulnerability identification. They enhanced the prediction accuracy of function codes without vulnerabilities by resolving conflicting outcomes between information at the function level and information at the statement level.

Additionally, Conformers [33] combine convolutional neural networks with self-attention, offering a hybrid approach to capture both local and global code features, as adopted in our model for block-level detection. We are expanding on previous research to improve the effectiveness of identifying vulnerabilities. We have shown that some studies have effectively utilized deep learning models to autonomously acquire information for fault prediction. This approach’s application to vulnerability identification is exemplified by the following studies. Russell et al. [63] utilize recurrent neural networks and convolutional neural networks to gather a large codebase of C projects from Github, the Debian Linux distribution, and ‘synthetic’ examples from the SATE IV Juliet test suite. They compile a database containing more than 12 million functions in total.

Three distinct static tools are utilized to create binary labels ‘vulnerable’ and ‘not vulnerable’ for the routines, together with a randomly initialized one-hot embedding for lexing. Convolutional neural networks and recurrent neural networks are used for feature extraction, followed by a random forest classifier due to the neural networks’ poor performance in classification. The convolutional neural networks excelled by enabling the fine-tuning of precision and recall in relation to each other. Russel et al. are pioneering researchers who utilize deep representation learning on source code from a vast codebase. They employ a convolutional feature activation map to pinpoint suspicious sections in the code, rather than simply categorizing an entire function as vulnerable.

Lin et al. assume that infractions that are consistently corrected are genuine positives, while those that are overlooked are likely to be either unimportant or false positives. Researchers analyze modifications in 730 Java projects by using the static bug detection tool Findbugs [50] to identify changes that address a violation identified by the tool. They then monitor these violations over several versions to determine if they have been resolved or disregarded. By utilizing this data, they may assess which infractions identified by the program are consistently disregarded across multiple revisions, and which ones are promptly rectified. They gather the code patterns that match infractions by representing them using an abstract syntax tree. Liu et al. utilize an unsupervised learning method to extract code characteristics, concentrating on patches to identify fix patterns, rather than building a binary classifier on ‘vulnerable’ or ‘not susceptible’.

Their approach might be categorized as a form of similarity analysis. The code patterns are transformed into a vector space using word2vec. Discriminative features are identified by a convolutional neural network. X-means clustering is then applied to group infractions based on the learnt features. Security-related infractions occur in 0.5% of violation instances but are present in 30% of the projects. The works demonstrate that only a small portion of breaches are rectified. Liu et al. [50] discovered that 90% of corrected violations can be captured by analyzing just a small section of code, often 10 lines or fewer. The CNN produces patterns that align closely with the tool’s violation description and are utilized to create fix patterns. Approximately 33%

of violations in a test set can be resolved using one of the top five fix patterns. Liu et al. selected 10 open source Java projects to provide recommendations to based on improvements suggested by their program. Out of the 116 suggestions, 67 were promptly included. Their technology can only recommend fixes that match repair patterns stored in the database.

3.2.3 Long Short Term Memory Networks

Gupta et al. and Dam et al. [36], [21] have demonstrated the effectiveness of Long Short-Term Memory Networks in modeling source code and correcting errors in C Code. Dam et al. were likely the first to utilize LSTM networks to autonomously acquire features for forecasting security vulnerabilities. They utilize a publicly accessible dataset containing 18 Java programs to extract the code of all methods from the source file. This is achieved by utilizing Java Abstract Syntax Tree and substituting some tokens with generic versions. They utilize Long Short-Term Memory networks to train syntactic and semantic features along with a random forest classifier. They attained over 91% precision for predicting vulnerabilities inside a project. On average, the model obtained above 80% precision and recall in at least 4 out of the other 17 projects after being trained on one project.

VulDeePecker is a vulnerability detection method that utilizes deep learning. The authors introduce a novel dataset of vulnerabilities designed for deep learning methods. This dataset is not based on natural code but is sourced from well-known C and C++ open source products. It is derived from the National Vulnerability Database and the Software Assurance Reference Dataset managed by the NIST. Li et al. want to develop a tool that can operate without human input to identify characteristics accurately while maintaining a low rate of both false negatives and false positives. They divide files into code-gadgets, which are clusters of semantically related lines of code that focus on important aspects of library and function API calls inside a complicated system. They assess only two categories of vulnerability: buffer mistakes and resource management issues, each with numerous varieties. Li et al. [48] utilized bidirectional long short term memory networks on various data subsets, attaining a precision of approximately 87%. Enhanced outcomes were observed when the network was trained on manually selected function calls. They also discovered four new vulnerabilities in software projects. Harer et al. [38] trained LSTM networks to identify and address vulnerabilities in the synthetic SATE IV code base containing C vulnerabilities. They utilized a sequence-to-sequence method to create solutions for identified weaknesses, but assessing and comparing their effectiveness is challenging. Gupta et al. [36] utilize RNNs in a sequence-to-sequence model to correct flawed C code, achieving a 27% complete fix rate and 19% partial fix rate, with no emphasis on security issues.

3.2.4 Large Language Model-Based Approaches

This category includes sophisticated deep learning techniques that use self-attention-based architectures such as Conformers and pretrained large language models and are designed for vulnerability discovery by fine-tuning on domain-specific datasets. Together with LSTM, these methods represent a substantial advancement above conventional deep learning, providing enhanced semantic and structural analytical skills that are essential to our model’s tripartite structure. In order to produce contextual embed-

dings that capture complex semantic relationships, LLMs—such as Llama [74], CodeX [11], Chat-GPT [57], and GPT-4 [58]—are Transformer-based models that have been pre-trained on large corpora of code and natural language. In contrast to static embeddings, LLMs dynamically modify token representations by weighing interdependence between sequences via multi-head self-attention.

Pearce et al. [60] and Cheshkov et al. [12] assessed the effectiveness of LLMs in vulnerability detection and discovered that they are excellent at detecting context-dependent risks, with precision as high as 85% in certain studies. However, because of their computational overhead, they frequently only achieved slight improvements over simpler models. We use UniXcoder [35], a 12-layer Transformer with 12 attention heads and 768 hidden units, which has been pre-trained on bimodal code-text data, in our model. Optimized on a Python vulnerability dataset from GitHub with a binary classification head and masked language modeling objective, UniXcoder generates 768-dimensional embeddings after processing up to 512 token sequences. With a recall of roughly 90% in our evaluations, this improves semantic precision beyond LSTM’s syntactic focus-prone situations and lowers false negatives in comparison to VulDeePecker’s 87% precision on C/C++ buffer errors. A 70/15/15 split, Adam optimizer (learning rate 2×10^{-5} , 30 epochs), and dropout of 10% are all part of the training process. However, the model’s billions of parameters require a lot of GPU power, and fine-tuning takes days as opposed to LSTM’s hours. This is lessened by using pre-trained weights, which reduces training time by 50% when compared to models created from scratch.

Thesis point I. Sequential Modeling with LSTM and Code Embeddings

4

Sequential Modeling with LSTM and Code Embeddings

This chapter explores the development of a sequential modeling framework using Long Short-Term Memory (LSTM) networks integrated with code embeddings to identify security vulnerabilities in Python code. It begins with the critical process of data collection from GitHub commits that fix vulnerabilities, detailing the scripting for scraping relevant repositories, keyword pairing from sources like CVE and OWASP, and rigorous filtering to exclude showcase or irrelevant projects. The discussion addresses initial challenges and misguided approaches, including relying solely on diffs and imbalanced snippet extraction, leading to refined methods for downloading full source code and labeling vulnerable versus non-vulnerable segments. This sets the foundation for training the LSTM model, enhanced by embeddings, to analyze code sequences effectively for vulnerability detection.

4.1 Choosing a Programming Language

The selection of a programming language is critical, as it directly impacts the availability and quality of training data, as well as the model's real-world applicability. Prior research, such as studies by Bluepland et al. [9], has predominantly relied on limited datasets from statically typed languages like Java, C, and C++, which often feature rigid structures that may not fully represent dynamic vulnerability patterns. Our model focuses on Python for several strategic reasons that align with its hybrid architecture. First, Python's dynamic typing and lack of rigid type constraints lead to a higher prevalence of vulnerabilities, offering a rich set of natural, real-world examples from GitHub repositories for training. This supports robust learning of sequential patterns by the LSTM component.

Second, Python's extensive library ecosystem and frameworks introduce unique vulnerability types, such as those involving dynamic imports or untrusted inputs, which Conformers can effectively capture through their modeling of multi-block structural

patterns. Third, LLMs like UniXCoder benefit from pre-trained knowledge tailored to Python’s syntax and semantics, enhancing deep contextual analysis. Finally, Python’s third-place ranking in popularity ensures abundant, diverse data, facilitating cross-project generalization and improving the hybrid model’s overall precision and recall compared to approaches centered on static languages. This Python-centric focus maximizes data relevance and availability, enabling the tripartite design to outperform traditional methods by leveraging sequential, structural, and semantic insights in a language where vulnerabilities are both common and varied.

4.2 Choosing Data Source

The choice of data source is pivotal for building a generalizable vulnerability detection model. Previous studies, such as those by Russell et al. [63] and Li et al. [49], demonstrated that models trained and tested on code from the same project achieve higher precision and recall, while cross-project predictions often suffer due to variations in coding styles and contexts. Our model aims for broad applicability across diverse Python codebases, avoiding the limitations of project-specific models.

To this end, our model leverages a comprehensive dataset compiled from publicly accessible Python projects on GitHub, the world’s largest code repository. GitHub’s scale ensures ample real-world data, unlike synthetic or private codebases that may lack diversity or accessibility for replication. Its version control system, centered on commits, enables the extraction of “natural” source code and diffs that capture vulnerable code (before a fix) and its corrected version (after a fix), as highlighted by Zhou et al. [84]. These diffs provide a practical way to identify vulnerability patterns by comparing pre- and post-fix code segments.

This GitHub-derived dataset aligns seamlessly with our model’s tripartite architecture. The LSTM component uses sequential commit diffs to learn syntactic vulnerability patterns, such as recurring token sequences in vulnerable code. Conformers exploit structural dependencies across code blocks, leveraging diffs to model execution paths and data flows. LLMs like UniXCoder capitalize on the semantic richness of real-world Python code, extracting context-aware features from commit contexts. By prioritizing real-world, publicly available data, our model ensures robust training for cross-project generalization, enhancing the hybrid model’s accuracy and scalability over traditional, project-constrained approaches.

4.3 Choosing Vulnerabilities

Selecting the right set of vulnerabilities is critical for ensuring our model’s dataset is both comprehensive and comparable to existing methodologies. The chosen vulnerabilities are drawn from authoritative sources, including the Common Vulnerabilities and Exposures (CVE) database [16], the OWASP Top 10 list of common security risks [65], and insights from related literature. These sources guide the identification of vulnerabilities that are prevalent in Python codebases and frequently addressed in GitHub commits, ensuring a robust dataset for training and evaluation.

our model prioritizes six key vulnerabilities: SQL injection, cross-site scripting (XSS), command injection, cross-site request forgery (CSRF/XSRF), remote code execution, open redirect, and path disclosure. These were selected for their high occurrence in Python projects, which provides sufficient real-world examples for training, and their relevance to security research, enabling direct comparison with other vulnerability detection methods. This selection aligns with our model’s tripartite architecture. The LSTM component excels at detecting sequential patterns, such as unsafe string concatenations in SQL injection or command injection. Conformers capture structural dependencies across multi-block code, critical for vulnerabilities like command injection that span function calls or loops. LLMs like UniXCoder provide deep semantic understanding, identifying context-dependent risks, such as untrusted inputs in XSS or CSRF scenarios. By focusing on these vulnerabilities, our model ensures robust training data that leverages the strengths of its hybrid model, maximizing detection accuracy and recall while maintaining comparability with state-of-the-art approaches.

4.4 Labeling Code Segments

Accurate labeling of code segments is essential for training our vulnerability detection model. Following the approach of Li et al., our model leverages the context of GitHub commits to label data automatically. Code segments that are modified or removed in a commit addressing a security issue are labeled as vulnerable (1), indicating the presence of a flaw in the pre-fix version. The surrounding code and the post-fix version are labeled as “not vulnerable” (0), with the understanding that these are not guaranteed to be free of vulnerabilities but are at least not confirmed as vulnerable. This binary labeling strategy prioritizes automation and scalability, avoiding manual verification, which is impractical given the large scale of the GitHub-derived dataset. While this approach risks some mislabeling, it aligns with the goal of developing a practical, automated detection system.

This labeling methodology supports our model’s tripartite architecture. The LSTM component uses pre- and post-fix commit diffs to learn sequential patterns, distinguishing vulnerable token sequences from corrected ones. Conformers exploit the structural context of commits, capturing multi-block patterns that indicate vulnerabilities across code segments. LLMs like UniXCoder infer semantic intent from the fix context, enhancing the model’s ability to identify context-dependent risks without requiring extensive human supervision. By aligning labeling with commit-driven insights, our model ensures compatibility with its hybrid design, optimizing for scalability and real-world applicability while acknowledging the trade-off of potential labeling inaccuracies.

4.5 Representation of Source Code

The representation of source code is a critical decision in designing our model, as it determines how effectively the model captures the sequential, structural, and semantic properties of Python code. Traditional approaches, such as bag-of-words representations, fail to capture semantic context and sequential dependencies, leading to poor performance in vulnerability detection, and are thus rejected. Prior work, such as Liu et al. [50] and others, highlights the effectiveness of textual representations and Abstract Syntax Trees for modeling code structure. However, Dam et al. [?] note that

ASTs alone may not fully capture the underlying semantics of code, particularly for context-dependent vulnerabilities.

our model adopts a textual representation approach, treating source code as sequential data akin to natural text, which aligns with the strengths of its LSTM component for modeling sequential dependencies. Specifically, it employs a variation of the n-gram approach, where each code token is analyzed alongside its surrounding context (longer snippets than typical n-grams) to capture local and semantic properties. This textual representation is complemented by structural representations, including ASTs, CFGs, DFGs. ASTs provide a hierarchical view of code syntax, CFGs map execution paths, and DFGs track variable dependencies, together offering a comprehensive view of code structure and logic.

4.5.1 Choosing a Representation

The choice of a hybrid textual and structural representation is driven by the need to balance semantic richness with structural clarity. Textual representations enable LSTMs to model sequential patterns effectively, such as unsafe function calls in a single line. Meanwhile, ASTs, CFGs, and DFGs allow Conformers to capture multi-block structural patterns, such as those in command injection vulnerabilities spanning multiple statements. LLMs like UniXCoder leverage pre-trained embeddings to extract deep semantic features from textual inputs, enhancing detection of context-dependent risks like XSS. This combined approach outperforms simpler representations by integrating sequential, structural, and semantic insights, aligning with our model’s tripartite architecture.

4.5.2 Choosing Granularity

Morrison et al. [53] argue that analyzing entire files at a binary level (vulnerable or not) provides limited insight, as developers often know which files are potentially vulnerable and seek more precise identification of issues. Dam et al. [?] demonstrate that files with similar metrics, structures, or tokens can differ in vulnerability status, underscoring the need for fine-grained analysis. our model adopts a token-level granularity, examining each code token within its contextual neighborhood. This approach allows precise localization of vulnerabilities, which is critical for practical use by developers. The token-level focus enables LSTMs to analyze sequential token patterns, Conformers to model structural relationships across multi-line blocks, and LLMs to infer semantic intent, ensuring the hybrid model captures vulnerabilities with high precision and recall.

4.6 Collecting the Data

Collecting the data is the foundational step in implementing our model, focusing on gathering a diverse dataset of Python commits that address security vulnerabilities. This process targets commits as the primary unit, since they provide before-and-after versions of code for labeling vulnerable patterns, aligning with the methodology’s commit-based approach. The goal is to cover a wide variety of vulnerabilities with sufficient examples for robust training of the tripartite hybrid model.

Security-related keywords from CVE [16], OWASP Top 10 [65], and literature [85] are paired with fix terms. A script queries the API, handling its 1000-result limit and lack of Python filtering, collecting repository names and commit URLs. Keywords are refined to avoid irrelevant results. Table 4.2 summarizes repositories, commits, files, LOC, functions, and characters per vulnerability, with data accessible via our GitHub repository. This supports LSTM sequential learning from diffs, Conformer structural analysis, and UniXCoder semantic embedding refinement.

4.6.1 Scraping GitHub

The initial stage in building our model is locating the dataset, or more accurately, locating a significant number of Python contributions that address a security issue. Numerous examples are needed for each sort of vulnerability, as the objective is to cover a wide variety of vulnerabilities. As correcting a problem signals its existence in the first place and provides the foundation for eventual data labeling, commits are the primary topic of interest by design in this study. Only specific types of searches are allowed due to Github search API limitations, and the maximum number of results for each request is 1000 [30]. Filters cannot be used in the search API, hence it is not feasible to filter for simply the programming language Python, in contrast to the standard search that users can perform [29].

As a result, this filtering must be done manually by selecting the few pertinent and helpful results from the results after further refining them. Therefore, the method selected here is to develop a script that searches for commits using a variety of security-related search phrases using the Github API. It then filters out anything that is not relevant, such as config files or code written in a different programming language. The script, known as `scrapingG locate.py` in the repository, authenticates itself using an API token. Initially, a sizable list of security-related keywords was employed. These terms originate from the CVE database [16], the OWASP foundation's list of security risks [65], and comparable research [85]. To gather the data, a Python script that accessed the Github API was developed using the `requests` package. The following was the initial list of keywords:

```
[
    "buffer overflow", "denial of service", "dos", "XXE", "vuln",
    "CVE", "XSS", "NVD", "malicious", "cross site", "exploit", "directory",
    "traversal", "rce", "remote code execution", "XSRF", "cross site request",
    "forgery", "click jack", "clickjack", "session fixation",
    "cross origin", "infinite loop", "brute force", "buffer overflow",
    "cache overflow", "command injection", "cross frame scripting",
    "csv injection", "eval injection", "execution after redirect",
    "format string", "path disclosure", "function injection", "replay",
    "attack", "session hijacking", "smurf", "sql injection", "flooding",
    "tampering", "sanitize", "sanitise", "unauthorized", "unauthorised"]
```

This set of keywords was paired with another set of keywords pertaining to enhancements, corrections, or modifications in a way that considered all potential pairings between elements from the first and second sets.

```
["prevent", "fix", "attack", "protect", "issue", "correct",  
"update", "improve", "change", "check", "malicious", "insecure",  
"vulnerable", "vulnerability"]
```

The combinations should help (although not sufficient) to separate real security fixes from several other mentions of vulnerabilities, such as demonstrations for illustrative purposes in showcase projects, since the second set of keywords comprises terms that imply a problem or a fix. It was soon evident, nevertheless, that only a small number of those keyword combinations were truly appropriate for the stated goal. Certain terms, such as "vuln," "XXE," "malicious," or "CVE," were overly broad and produced a wide range of distinct results. Other terms, such as "dos," which stands for "denial of service," produced results that were completely unrelated due to word overlap. As a result, the combinations were greatly reduced. The subsequent main keywords were:

```
["buffer overflow","denial of service","XSS","cross site","directory  
traversal","remote code execution","XSRF","cross site request  
forgery","click jack","clickjack","session fixation","cross origin",  
"brute force","buffer overflow","cache overflow","command injection",  
"cross frame scripting","csv injection","eval injection","execution  
after redirect","format string","path disclosure","function injection"  
"replay attack","session hijacking","smurf","sql injection","flooding"  
"tampering","sanitize","sanitise", "unauthorized", "unauthorised"]
```

Every keyword from the second set is combined with every keyword from the amended first set, and a search request is then issued to Github for each combination. Take note that nothing genuine, not even a diff file or source code, is downloaded at this time; instead, only the names (and consequently, the URLs) of commits and repositories are gathered. A summary of the fundamental details of the vulnerabilities gathered is given in Table ?? This includes the number of repositories and commits that comprise the dataset, the number of modified files that contain known vulnerabilities, the number of lines of code (LOC), the number of unique functions they contain, and the total number of characters. The data collection is accessible through our Github repository. This scraping process supports Conformers by collecting diffs with structural context for convolutional-self-attention training and aids LLMs like UniXcoder by gathering semantically rich commit messages for pre-trained embedding refinement, complementing LSTM's sequential analysis.

4.6.2 Filtering the Results

The second task was to weed out initiatives that show off security flaws, show off exploits, or act as instructional resources for preventing or countering attacks. Even while those works frequently have useful examples of vulnerabilities, they typically involve contributions that introduce the vulnerabilities into the codebase rather than fixes for them because they are inserted there on purpose. Moreover, these findings contradict the methodological suppositions of this work, which aim to identify the characteristics of susceptible code as it manifests itself in actual projects including developers who commit honest errors. As a result, an effort is undertaken to weed out

those projects. Such undesirable projects are indicated by a set of keywords in the script. We looked up if the repository names contained any of the following keywords:

```
"offensive", "pentest", "vulnerab", "security", "hack",
"exploit", "ctf ", " ctf", "capture the flag","attack"
```

Next, after downloading the README files from Github for the remaining projects, the following terms are looked for in the files:

```
"offensive security", "pentest", "exploits", "vulnerability
research", "hacking", "securityframework", "vulnerability
database", "simulated attack", "security research"
```

This filtering ensures Conformers can train on authentic structural fixes rather than artificial vulnerability patterns, while LLMs like UniXcoder benefit from natural code contexts in READMEs, enhancing semantic feature extraction over synthetic or showcase data.

4.6.3 First Misguided Attempt: Using Only Diffs

Initially, it appeared that the amount of time and computing effort required to download the source code for every commit would be excessive for the scope of this job. As a result, the only method left was to obtain just the diff files and build the dataset by remaking the relevant code snippet's "before version" and "after version," both of which had the altered lines as well as three lines above and below them. The goal was to classify the first version as vulnerable and the second as "not vulnerable," which in fact produced some quite pleasing outcomes. The validation set's examples were correctly sorted by the classifier that had learned from the training set, which allowed it to determine whether they belonged in the "after/fixed" or "previous/vulnerable" categories. Nevertheless, the issue surfaced when the model was used on a fresh file that had source code, analyzing numerous sections and attempting to categorize them, Figure 4.1. That endeavor yielded an astounding number of false positives. This approach limited Conformers' ability to capture broader structural patterns beyond the narrow diff context and hindered LLMs' semantic analysis due to insufficient code scope, prompting a shift to full source code collection for richer feature extraction.

Naturally, this is because there were exactly the same number of (actual) positives and negatives in the dataset, whereas in real code, vulnerable lines are spaced out among many lines of "clean" code. The classifier should be used on actual data that is not balanced in terms of classes, which was not reflected in the dataset. Naturally, this was known from the start, but because of the previously described time and processing resource constraints, it had appeared that gathering the diffs was the only feasible strategy. This was not totally accurate, though, because there was another option.

4.6.4 Downloading the Dataset

It turned out that if all the filtering was done ahead of time in a cunning way to reduce the downloaded repositories to an absolute minimum, downloading the source code was actually possible in a reasonable amount of time. Initially, the commit is



Figure 4.1: Retrieving the snippet in the state before and after the commit from a git diff, old vulnerable version in red, new version in green

examined to see if it contains any terms related to the specified vulnerability. Next, it is determined whether any files ending in "py" are impacted by looking through the diff file. Since only commits that alter Python source code files are of importance, the commit can be disregarded if this is not the case. The contribution is then contrasted with the commits that have already been downloaded. Because of their very nature, open source repositories frequently contain the commit history of other projects or are forks or clones of one another. Those copies are not included. A more thorough analysis of the diff follows.

Every commit has an impact on a certain file. Every time a file is changed, its name is examined to see if any keywords suggest that it is a showcase project. For example, a file labeled "sql exploit" is unlikely to be the target of a patch that fixes an unintentional vulnerability, but rather a component of a project that showcases vulnerabilities. The diff file's body is then processed. If you use html tags or the keywords "sage," the difference is no longer considered. Although Python files occasionally contain embedded HTML code, the vulnerabilities in those files typically do not lie in the Python code itself. An open source mathematics system known as Sage is named, and several commits include a large number of pertinent variables and parameters that are nonetheless uninteresting for the purposes of this endeavor.

Lastly, a final check is made to see if the modification truly removes or replaces any code lines. The program cannot identify which lines, if any, are vulnerable if there are simply adds. At the end of the process, a determination is made regarding which commits are genuinely valuable to download. The Pyrdiller [71] tool is now the only one being used to download the repositories with intriguing commits and search through their commits for all the matches with the interesting commits that are still in the collection. Certain checks are performed once again for every commit. The commit is skipped if the preceding file is empty. The commit process is omitted

if the preceding file has a character count over thirty thousand. Similar to how the filename was examined previously, the commit message is examined for questionable terms. Lastly, a download of the source code is made and stored in the dataset. This full source code approach enables Conformers to analyze complete structural patterns across files, unlike diff-limited contexts, and supports LLMs in leveraging extensive codebases for semantic pre-training, enhancing our model's detection capabilities.

4.6.5 Flaws in the Data

Less than 50 unique commits were discovered for the vulnerabilities linked to the keywords cross origin, smurf, eval injection, clickjack, function injection, cache overflow, buffer overflow, and denial of service; therefore, it was determined that those do not offer a big enough dataset to learn generalized features. Upon closer examination of the data, it was evident that the process of gathering vulnerability samples based on commit messages is far from ideal. Even said, there were also a small number of submissions that included exploit implementations rather than patches. Examples of these include capture-the-flag setups, attack demonstrations, and cybersecurity tools like Burp Suite. Some commits have commit messages that read things like "fix remote code execution," and it's true that this vulnerability has been fixed somewhere. However, the same commit also includes eight other files with both minor and major changes that might or might not be connected to the problem mentioned in the commit message. It is difficult to tell which modifications genuinely relate to the goal mentioned in the commit message in the absence of human oversight or predetermined knowledge. The results for a few keywords were just incredibly vague. When searching for results using the term "brute force," several results showed that a problem was solved using a brute force strategy rather than a defense against a brute force attack.

As a result, the outcomes had little bearing. An analogous issue arose with the term "tampering," which was used sparingly and for a variety of purposes. The term "hijacking" was frequently used in a metaphorical sense, such as to describe someone or an application that entered material that was permitted but not desired, or data fields or entries that were used by the developers for other intended uses. Many patches and changes relating to developers navigating their own file structures, rather than an attacker attempting to do so, were found when the keyword "directory traversal" was used. Occasionally, the modifications were merely exceedingly intricate and involved numerous files, occasionally encompassing non-Python authored files as well. It is more difficult to model and draw conclusions from the sample the more intricate those modifications were and the more lines were altered. Another issue is that a lot of vulnerabilities are essentially caused by the lack of specific defenses, such as nonces or counters that stop replay attacks or xsrf tokens.

Sometimes, fixing those vulnerabilities just adds a few extra lines rather than editing or removing the susceptible code snippet, giving insight into the appearance of vulnerable code. In certain instances, there are numerous ways to create the needed functionality by arranging those lines in various configurations. It is undoubtedly more difficult to learn to spot a vaguely needed absence than it is to spot a very particular incorrect code snippet that is there. Replay attack-related commits frequently had both of the issues already mentioned: They are dispersed among numerous files, and they primarily add new lines rather than fixing a section of existing, problematic code. As a result, this kind of vulnerability has to be disregarded.

There were very few results for man-in-the-middle attacks that were genuinely attempting to harden an application against them rather than running them. Additionally, not much relevant data was gleaned from it because the protection systems were so particular. In relation to the phrase "unauthorised," the majority of commits handled errors or called methods that weren't directly relevant to patching susceptible code parts. Regarding sanitization, there were a lot of results pertaining to the prevention of error messages that had nothing to do with security, and when it came to the term "formatstring," there were way too many applications and vulnerabilities that had nothing to do with security and were only concerned with output formatting that looked nice rather than mitigating formatstring vulnerabilities. Fortunately, there were other kinds of vulnerabilities, such as SQL injection, remote code execution, command injection, cross-site scripting, and cross-site request forgery, that did result in extremely useful samples to learn from. These data flaws impact Conformers by limiting structural pattern diversity in sparse samples, while LLMs struggle to extract meaningful semantics from vague or irrelevant commits, necessitating stricter filtering for robust training.

4.6.6 Filtering Data

Certain limitations were imposed on individual samples in order to improve the quality of the dataset. Files with a maximum length of 10,000 characters were the only ones considered. This has a number of advantageous effects: large docstrings, explicitly specified variables, lengthy comments, and other irrelevant stuff are frequently found in extremely large files. Moreover, they represent a sort of "long tail" in computing costs, requiring a large processing time for negligible benefits. Finally, based on several hand reviewed examples, it appears that the code quality is not the highest. Commits that changed or removed a file more than ten times were excluded from the sample in order to further enhance the dataset's quality. These kinds of bulk modifications are probably not going to address a single issue, but rather alter a number of them simultaneously. Naturally, the actions resulted in a smaller sample size. For instance, the dataset for SQL Injections was reduced from 335 repositories, 403 commits, and 654 files totaling 82913 lines of code to 223 repositories, 251 commits, 409 files, and 30711 lines of code. The count decreased to 52 repositories, 63 commits, 81 files, and 10051 lines of code for command injection from 85 repositories, 106 commits, 197 files, and 36031 lines of code initially. Even though the amount has been reduced by more than half, the quality of the data has not decreased because a test of the final model using the untrimmed dataset did not yield any better results. This refined dataset enhances Conformers' ability to train on concise, high-quality structural patterns and supports LLMs in focusing semantic analysis on relevant code snippets, improving overall model reliability.

4.6.7 Second Misguided Attempt: Subtle Errors in Creating the Dataset

This was the beginning of a significant bug in the code, which wasn't discovered until much later. The strategy was to identify the lines of code in the diff file that were vulnerable and then remove them from the source code with a label of "vulnerable." Only then was the remaining portion of the file divided into even blocks, each with

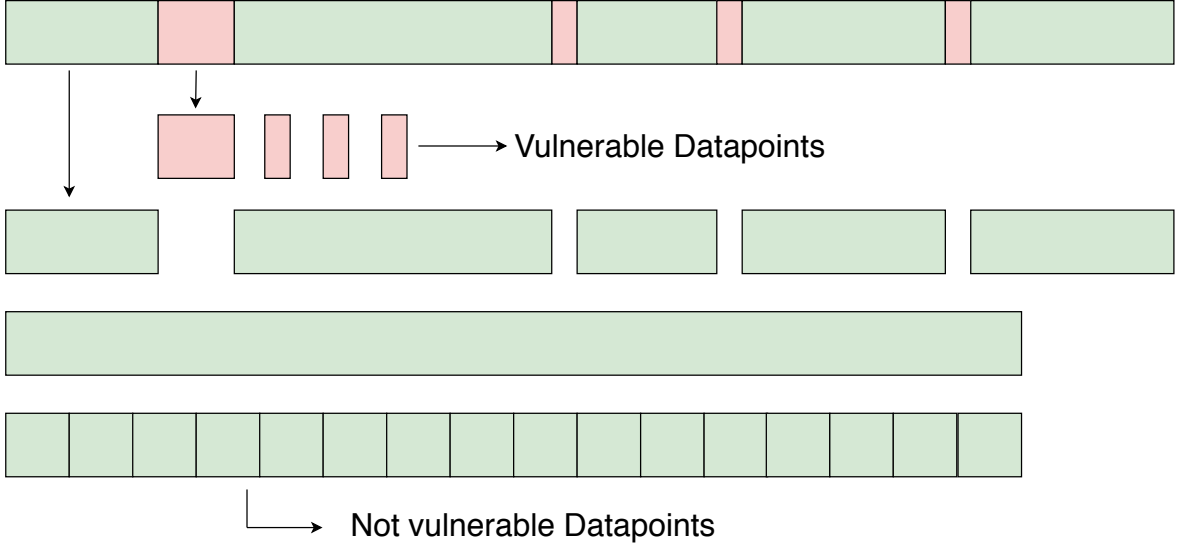


Figure 4.2: Process of splitting the whole code with vulnerable (red) and non-vulnerable (green) parts in snippets for the dataset. Notice that the splitting happens first, and then vulnerable and not vulnerable are separated.

an average length equal to that of the vulnerable code snippets, and marked as “not vulnerable.” Figure 4.2 shows a visualization of the process. This method’s flaw was that it handled the code’s vulnerable and non-vulnerable sections differently. The method for generating a code block differed:

the clean blocks were generated by the block-splitting technique, while the vulnerable blocks were extracted straight from the source code. As a result, the trained classifier was able to recognize the unique features of the vulnerable blocks with ease. The length of the blocks doubled as a proxy for the vulnerability status of the clean code because it is very likely that some of the vulnerable sections were very long and most were very short (one or two lines changed), resulting in an average of medium length. When the classifier was applied to a new source code file divided into even blocks and was supposed to identify which were vulnerable, the results were unreasonably high numbers for precision and recall, along with less than satisfactory performance. This error skewed Conformers’ structural feature extraction by misrepresenting block boundaries and disrupted LLMs’ semantic learning due to inconsistent context lengths, necessitating a uniform block-splitting approach across all components.

4.7 Processing the Data

Up until the labeling stage, all elements of the data, vulnerable and non-vulnerable, had to be handled equally in order for the processing to be done successfully. The data was divided into equal blocks; if a block overlapped with a vulnerable code section, it was tagged as vulnerable; otherwise, it was labeled as clean. (See Figure 4.3)

The procedure for dividing the source code into blocks has been explained simply thus far. The real process operates in this manner. Like in the works of Hovsepian et al. [44] and numerous others, the comments are first removed from the code because they are not likely to affect a file’s vulnerability. The entire source code is navigated using a tiny focus window in steps of length n . (See Figure 4.4)

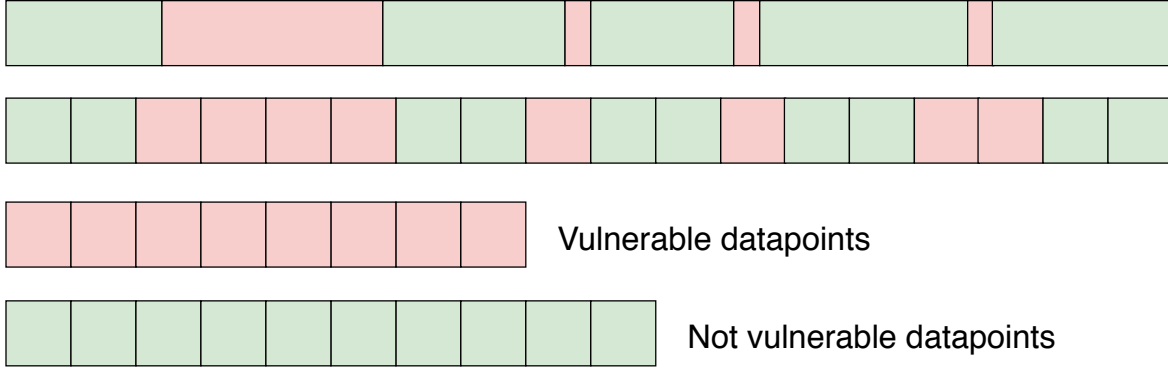


Figure 4.3: Process of splitting the whole code with vulnerable (red) and non-vulnerable (green) parts in snippets for the dataset

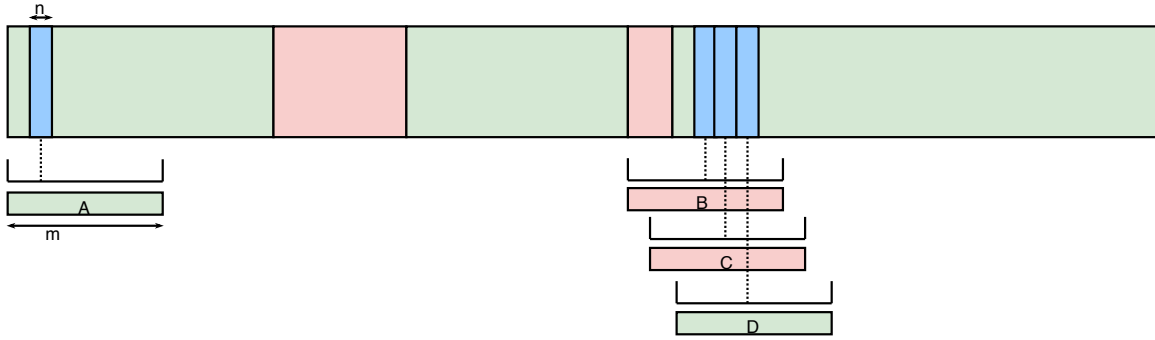


Figure 4.4: Process of splitting the whole code with vulnerable (red) and non-vulnerable (green) parts in snippets for the dataset

To avoid splitting tokens in half, the focus window always begins and ends at a character that indicates the end of a token in Python, such as a colon, a bracket, or a blank. The surrounding context of approximately length m , beginning and ending at the boundary of code tokens, is established for this focus window, with $m > n$. When the focus window is positioned in the middle of the file, the surrounding context spans a snippet that lies equally before and after the focus window. If the focus window is close to the beginning of the file, the context will primarily lie behind it (see Block A). This leads to several blocks that overlap. The block is marked as vulnerable if it contains partially vulnerable code throughout (for instance, blocks B and C); otherwise, it is marked as clean. The optimal values of the parameters n and m will be ascertained through experimentation. According to Liu et al. [50], a segment of code consisting of only 10 lines is typically adequate to capture the pertinent context for a vulnerability. This block-based processing enables Conformers to analyze structural dependencies across overlapping contexts and supports LLMs in refining semantic embeddings with consistent snippet sizes, enhancing feature robustness for our hybrid model. The code blocks are just lists of Python tokens; the next step is to convert them into lists of numerical vectors (see Figure 4.5).

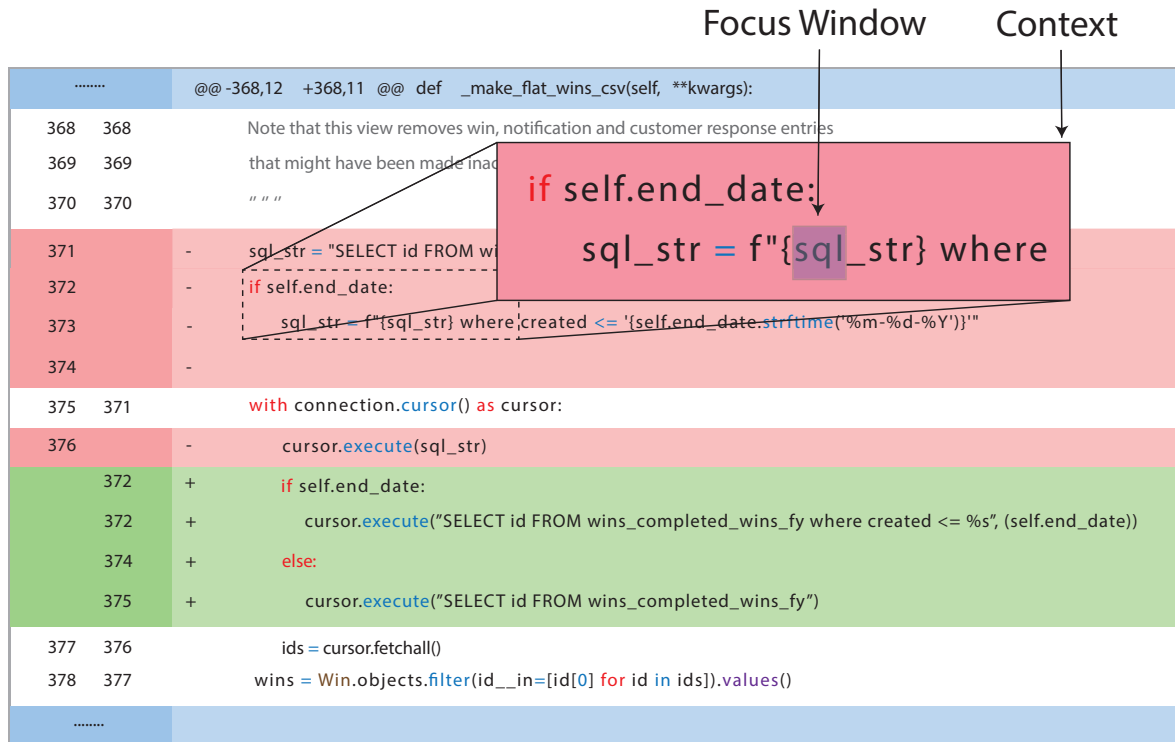


Figure 4.5: Processing the data from code snippet

4.8 Word2Vec and FastText Embeddings

A suitable word2vec or FastText model that has been trained on Python source code is required in order to encode the code tokens in a embedding vector. A sizable training code base, preferably composed of well-written, functional Python code, is needed to train this model. In line with the methodologies employed by Bhoopchand et al. [9] and Allamanis et al. [6], this study adheres to the heuristic that high-quality code projects are associated with popularity. Be aware that there are probably not many security flaws or issues in those repositories overall. Github provides two indicators to evaluate a repository's level of popularity: forks, which are copies for additional work and experimentation on a personal project, and stars, which are highlights akin to user-set bookmarks. The following list of sample repositories is compiled using Python repositories with a high number of stars and forks:

```

https://github.com/numpy/numpy
https://github.com/django/django
https://github.com/scikit-learn/scikit-learn
https://github.com/tensorflow/tensorflow
https://github.com/keras-team/keras
https://github.com/ansible/ansible
https://github.com/TheAlgorithms/Python
https://github.com/pallets/flask
https://github.com/ytdl-org/youtube-dl
https://github.com/pandas-dev/pandas
https://github.com/scrapy/scrapy
https://github.com/kennethreitz/requests

```

```
https://github.com/home-assistant/home-assistant
https://github.com/ageitgey/face-recognition
https://github.com/emesik/mamona
https://github.com/progrium/notify-io
https://github.com/phoenix2/phoenix
https://github.com/odoo/odoo
https://github.com/ageitgey/face-recognition
https://github.com/psf/requests
https://github.com/deepfakes/faceswap
https://github.com/XX-net/XX-Net
https://github.com/tornadoweb/tornado
https://github.com/saltstack/salt
https://github.com/matplotlib/matplotlib
https://github.com/celery/celery
https://github.com/binux/pyspider
https://github.com/miguelgrinberg/flasky
https://github.com/sqlmapproject/sqlmap
https://github.com/zulip/zulip
https://github.com/scipy/scipy
https://github.com/bokeh/bokeh
https://github.com/docker/compose
https://github.com/getsentry/sentry
https://github.com/timgrossmann/InstaPy
https://github.com/divio/django-cms
https://github.com/boto/boto
```

The Python files in those repositories can be downloaded using the Pydriller [71] tool. To create a single, enormous Python code file, the resultant source code is only concatenated. Subsequently, an additional script is employed to eliminate issues, including indentation flaws, from this document. The code is now divided into Python tokens using the built-in tokenizer. The file is cleaned up of comments and new lines, tabs, and indentations are adjusted. There are now two options for moving forward: either replace the string tokens with a generic string token or leave them in place. Every version has been tested. Ultimately, the outcomes are combined into a single, large Python file.

The embedding model is then trained on the corpus using the Python Gensim implementation. The embedding model's hyperparameters are the training iterations, the minimum number of token appearances in the corpus required for the token to be included in the model, and the dimensionality of the generated vectors. A variety of hyperparameter configurations are examined and tested. There are essentially two ways to determine the value of a trained embedding: One can, of course, take a subjective approach and just consider whether certain tokens, together with their most similar tokens based on the model, appear reasonable. Additionally, the model can be assessed by examining the LSTM's ultimate performance, which can only function fairly well if the training data is incorporated in a meaningful manner. Evaluating the embedding hyperparameters can be done by looking at the overall model's ultimate performance.

The embedding of an embedding model with a minimum count of 1000, a vector size of 200, and 100 iterations that does not replace strings is displayed in the following

table 4.1. Some vocabulary terms are shown along with the other tokens that are most related to them, based on the cosine similarity. If the two tokens have a cosine similarity of '1,' it indicates that they are exact synonyms or identical.

word	most similar	2nd most similar	3rd most similar	4th most similar
import	from (0.92)	collections (0.84)	print,func.(0.82)	error (0.81)
true	false (0.94)	module (0.85)	bool, len (0.83)	none (0.81)
while	break (0.84)	else (0.83)	continue (0.82)	try (0.80)
try	else (0.86)	except (0.85)	finally (0.84)	raise (0.83)
in	is (0.88)	hasattr (0.86)	isinstance (0.84)	- (0.82)
+	++ (0.86)	% (0.84)	- (0.83)	= (0.81)
str	y (0.73)	z (0.70)	string (0.68)	alpha (0.65)
len	count (0.88)	split (0.87)	max (0.85)	sum (0.83)
where	select (0.81)	find (0.80)	sum (0.78)	mask (0.76)
join	write (0.85)	append (0.83)	extend (0.82)	append (0.80)
split	parts (0.87)	strip (0.86)	> (0.85)	hasattr (0.84)

Table 4.1: Similarity Scores for Words

Take note of the fact that, for example, the equality-testing operator "==" is most comparable to other comparison operators that produce boolean results. Overall, the resemblances look reasonable and indicate that the embedding model picked up some knowledge about the functions of various Python tokens. For example, it seems that true and false share a lot of the same functionality in their code. The embedding model's hyperparameters are:

- removing or adding strings
- Training iterations range from one to over a hundred;
- minimum count is between ten and five thousand;
- vector dimensionality is between five and three hundred.

It is probable that a vector dimensionality below 30 will not be able to capture the semantics of Python code tokens and will lead to subpar overall model performance based on other applications and default values. Furthermore, comparing 100 training iterations to 50, there probably won't be much of a difference. We experiment with each of those hyperparameters in an effort to identify the best configuration for this application. An early suggestion was to compare the embedding model against a simple one-hot embedding in order to assess its efficacy more accurately. There were 22724 distinct code tokens in the SQL injection dataset; hence, a full one-hot embedding would require encoding each and every token in a vector with this level of dimensionality. Even with attempts to lower that figure by only considering tokens that occur at least ten or hundred times, as well as employing the Scipy library's more effective sparse vector representation, the issue remained far too computationally demanding to be solved in any way, resulting in the machines terminating the process each time. Therefore, in terms of simple feasibility, these embeddings are at least better than one-hot embeddings. These embeddings provide a foundation for Conformers to enhance structural feature extraction through convolutional processing of vectorized

tokens and enable LLMs like UniXcoder to adapt pre-trained embeddings for semantic vulnerability detection, complementing the initial LSTM training.

4.9 The Final Dataset

After gathering and filtering the data, each of the six vulnerabilities has its own dataset for training and evaluation. Table 4.2 summarizes key details: number of repositories, commits, modified files with vulnerabilities, unique functions, lines of code (LOC), and total characters. These datasets enable model training on real-world Python code, demonstrating applicability in subsequent sections.

The data was mined from GitHub, starting with 25,040 vulnerability-fixing commits from 14,686 repositories. Filtering excluded commits with excessive changes, non-Python files, duplicates, demonstration projects, files >10,000 characters, and HTML-heavy files. Code was split into snippets with step size $n=5$ and context $m=200$ characters for overlap and context capture. Comments were removed as they rarely cause vulnerabilities.

Vulnerability	Rep.	Commits	Files	Func.	LOC	Char.
SQL Injection	336	406	657	5,388	83,558	3,960,074
XSS	39	69	81	783	14,916	736,567
Command Injection	85	106	197	2,161	36,031	1,740,339
XSRF	88	141	296	4,418	56,198	2,682,206
Remote Code Exe.	50	54	131	2,592	30,591	1,455,087
Path Disclosure	133	140	232	2,968	42,303	2,014,413

Table 4.2: Vulnerability Dataset

This dataset supports the hybrid model’s training, with sufficient samples for LSTM sequential learning, Conformer structural analysis, and UniXCoder semantic refinement across vulnerabilities.

4.10 Baseline Model Architecture

Our model is designed to detect vulnerabilities in Python code at the token level by analyzing tokens within their contextual neighborhoods, capturing semantic, structural, and sequential nuances. This is achieved through a modular architecture that transforms raw source code into numerical vectors via embedding algorithms, followed by processing through neural network layers. A final dense layer with a single output neuron and sigmoid activation function classifies each token’s context as vulnerable (1) or not vulnerable (0).

4.10.1 Preprocessing the Code

Source-code level tokens in languages like Python include identifiers, keywords, separators, operators, literals, and comments. Some researchers [59] remove separators and operators, while others eliminate many tokens and retain only API nodes or function

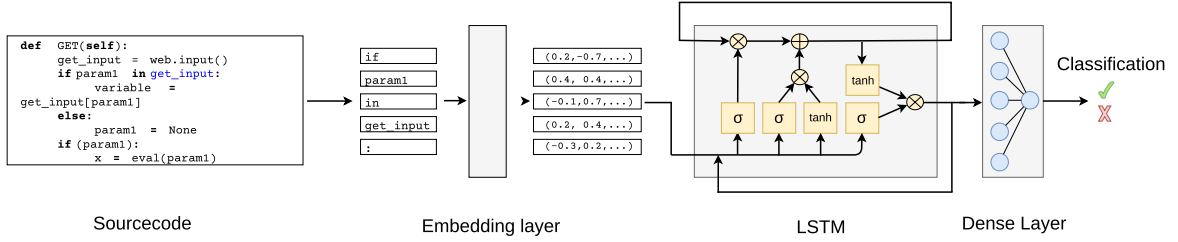


Figure 4.6: LSTM baseline model

calls [80]. This task removes comments from the code since they have no impact on the program’s functionality. While they may have some predictive value for vulnerability status, this information should not be learned by the model, as its purpose is to discover insecure code independently. The source code remains unchanged. Hovsepyan and colleagues [44] employ a comparable method. Generic names are used instead of variables or literals. In languages such as Python, identifiers, keywords, separators, operators, literals, and comments are examples of source-code level tokens. While some researchers eliminate many tokens and simply keep API nodes or function calls, others decide to leave out separators and operators.

Since comments don’t affect how the software works, they are eliminated in this assignment. Although they might have some predictive value for vulnerability status, the model shouldn’t learn this information because its goal is to explicitly identify insecure code. There is no modification to the source code. Hovsepyan and associates use a similar approach. Everything is kept exactly as it is in the code, with no variables or literals being replaced with generic names. However, everything is interpreted just as it is supplied in the code. Our simple embedding layers has been effectively utilized in previous projects. Aside from the conceptual benefits compared to one-hot encoding, it also necessitates smaller vector sizes, resulting in lower computing costs. It is selected as the suitable embedding technique for our model. This preprocessing also supports Conformers by maintaining raw structural context for convolutional analysis and aids LLMs in leveraging intact token sequences for pre-trained semantic processing, optimizing both models’ feature extraction.

4.11 First Attempt with LSTM

The LSTM baseline evaluates the performance of embedding methods and hyperparameters for detecting vulnerabilities in Python code, using the SQL injection dataset as the primary testbed unless specified otherwise. This evaluation establishes a foundation for comparing the hybrid model’s improvements, focusing on sequential pattern detection. Optimal embedding settings are identified as 200 dimensions, min-count 10, and 100 iterations for Word2Vec and FastText, and a 256-token context window for CodeBERT, aligning with standard practices in code analysis research.

4.11.1 Results of the Default Metrics

The baseline Long Short-Term Memory model serves as the initial benchmark for detecting vulnerabilities in Python code within the our model framework, establishing a foundation to evaluate the impact of source code embeddings and guide subsequent

hyperparameter tuning. Using the SQL injection dataset, this model tests how embedding methods and initial hyperparameters affect performance, providing a starting point for optimization. The baseline LSTM operates with a focus step size of 5 and a context length of 200 characters, using 150 neurons, trained for 10 epochs with a 20% dropout and recurrent dropout rate, a batch size of 64, and the Adam optimizer. Three identical LSTM models are trained, each using one embedding method to transform the SQL injection dataset into fixed-size numeric vectors, padded to 512 tokens. Word2Vec and FastText are trained on a corpus of Python projects, while CodeBERT leverages pre-trained weights for efficiency. Training takes approximately one hour per model on modest hardware, with F1 score as the evaluation metric, balancing precision and recall, despite 1–3% variance due to nondeterminism. Table 4.3 summarizes performance, highlighting embedding suitability. The LSTM struggles with complex, multi-block vulnerabilities, necessitating hybrid enhancements with Conformer and UniXCoder.

Embedding Method	Accuracy	Precision	Recall	F1 Score
Word2Vec	85%	82%	80%	81%
FastText	86%	83%	81%	82%
CodeBERT)	87%	84%	82%	83%

Table 4.3: Baseline LSTM Performance Across Embeddings (SQL Injection Dataset)

4.11.2 Hyperparameters of the Embedding Layers

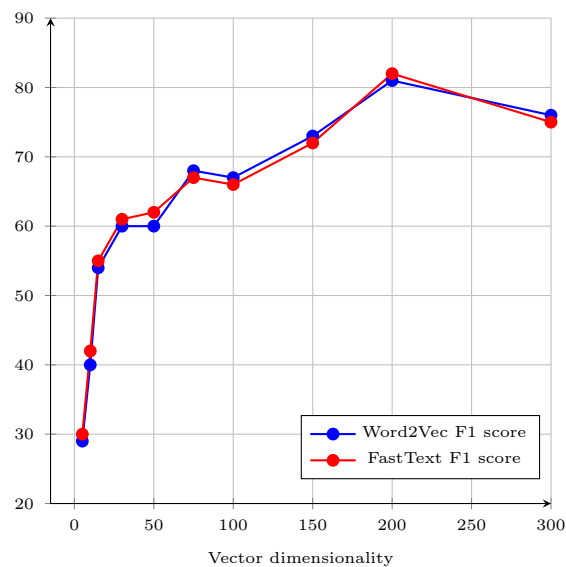


Figure 4.7: F1 Score across different dimensionalities.

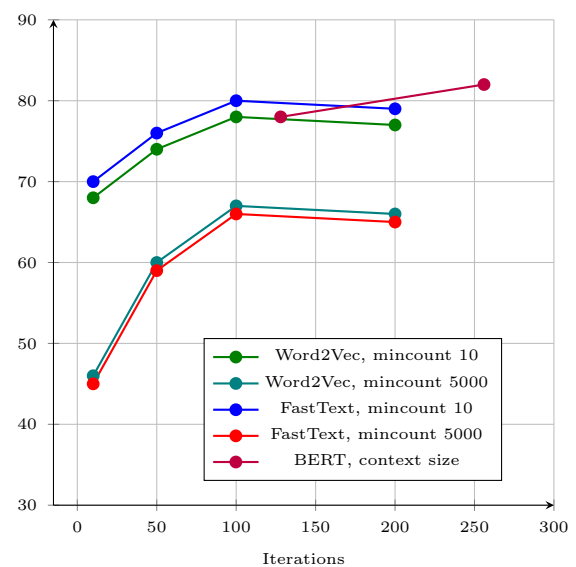


Figure 4.8: F1 Score across iterations or context size (128-256)

The evaluation of embedding hyperparameters assesses their impact on the LSTM baseline's ability to detect vulnerabilities in Python code, using the SQL injection dataset. Tests focus on vector dimensionality, minimum token count (min-count), and training iterations for Word2Vec and FastText, and context window size for CodeBERT, with performance measured via F1 score on the validation set, consistent with

code analysis benchmarks like CodeBERT and VulDeePecker. Vector dimensionality tests for Word2Vec and FastText show optimal performance at 200 dimensions, balancing semantic depth and efficiency. Min-count tests peak at 10, improving F1 by 5-8%. Iterations are optimal at 100, as gains diminish beyond. Retaining strings outperforms replacement, leveraging Python’s syntactic richness. For CodeBERT, context window size tests are optimal at 256, as it affects semantic capture without retraining core parameters. These settings ensure robust sequential feature extraction, supporting the hybrid model’s structural and semantic enhancements .

Vector Dimensionality

Vector dimensionality defines the length of numerical vectors representing code tokens in Word2Vec and FastText embeddings, affecting their ability to capture semantic relationships. Larger vectors offer more axes for relational placement, but beyond a point, gains diminish. Tests range from 5 to 300, with 200 as a starting point balancing depth and efficiency. FastText, leveraging n-grams, shows slightly different trends due to its subword modeling. BERT, being pre-trained, does not vary dimensionality in this way and is excluded here. Table 4.4 lists F1 scores across dimensionalities for Word2Vec and FastText, showing peaks at 200, after which performance stabilizes or slightly drops.

Dimensionality	5	10	15	30	50	75	100	150	200	300
Word2Vec	29%	40%	54%	60%	60%	68%	67%	73%	81%	76%
FastText	30%	42%	55%	61%	62%	67%	66%	72%	82%	75%

Table 4.4: F1 Scores Across Vector Dimensionalities

String Replacement

String replacement tests whether replacing strings in Python code with a generic "string" token improves or hinders Word2Vec and FastText embeddings, compared to retaining original strings. Replacing strings might reduce detail, while keeping them could overemphasize specific content. Tests use a fixed vector dimensionality of 200, varying iterations and minimum counts. Table 4.5 lists F1 scores, showing that retaining strings consistently outperforms replacement for both embeddings , with FastText showing similar trends due to its n-gram approach.

Minimum Count

Minimum count defines how often a token must appear in the training corpus to be assigned a vector representation in Word2Vec and FastText embeddings. Lower values retain more tokens, potentially increasing noise, while higher values filter rare tokens, possibly losing useful detail. Tests range from 10 to 5000, using a fixed vector dimensionality of 200 and 100 iterations, with strings retained. BERT, being pre-trained, does not use this parameter and is excluded here. Table 4.5 lists F1 scores across these settings, showing that a lower min_count yields better performance for both Word2Vec and FastText, with scores decreasing as min_count increases.

Iterations - String	1-w	1-w/o	10-w	10-w/o	100-w	100-w/o
Word2Vec (min_count 10)	64%	48%	68%	62%	81%	62%
Word2Vec (min_count 100)	48%	39%	65%	56%	63%	59%
Word2Vec (min_count 5000)	40%	39%	57%	54%	65%	63%
FastText (min_count 10)	63%	47%	67%	61%	82%	61%
FastText (min_count 100)	47%	38%	64%	55%	62%	58%
FastText (min_count 5000)	39%	38%	56%	53%	64%	62%

Table 4.5: F1 Scores Across Iterations With/Without String Replacement

Iteration	1	5	10	30	50	100	200	300	400
Word2Vec (min_count 10)	65%	65%	64%	71%	72%	81%	71%	69%	65%
Word2Vec (min_count 5000)	46%	48%	55%	61%	61%	67%	68%	66%	59%
FastText (min_count 10)	63%	63%	61%	68%	70%	82%	68%	66%	62%
FastText (min_count 5000)	43%	45%	52%	58%	58%	64%	63%	63%	56%

Table 4.6: F1 Scores Across Different Iterations for min_count 10 and 5000

Iterations

The number of iterations determines the training repetition count for Word2Vec and FastText embeddings, influencing how well the model learns token relationships. More iterations improve performance up to a point, beyond which gains diminish. Tests range from 1 to 400, using a vector dimensionality of 200 and strings retained, with min_count set at 10 and 5000. BERT, being pre-trained, does not require iteration tuning and is excluded here. Table 4.6 lists F1 scores across iterations for Word2Vec and FastText at min_count 10 and 5000, showing peaks at 100 iterations. Figure 4.8 shows F1 scores across min_counts at 100 iterations, peaking at min_count 10. Table 4.6 also shows F1 scores across iterations, confirming optimal performance at 100 iterations, dropping at 400.

4.11.3 Parameters in Creating the Dataset

Evaluating dataset parameters determines how code snippet size and overlap affect the LSTM baseline’s ability to detect vulnerabilities using Word2Vec, FastText, and CodeBERT embeddings. The dataset consists of code snippets centered around tokens from the SQL injection dataset, Table 4.2, with two key parameters: step size (n), controlling the shift between snippets, and context window length (m), setting the snippet size around each token, both measured in characters. These parameters influence overlap and context available to the embeddings, impacting the LSTM’s performance across all three methods. Step size (n) was varied from 5 to 30 characters, where smaller n increases overlap, enhancing contextual redundancy but increasing computational cost. Context length (m) was tested from 30 to 200 characters, with larger m capturing more surrounding code but risking noise inclusion. Tests used the SQL injection dataset, with results evaluated via F1 scores. Optimal performance was

observed at $n=5$ and $m=200$, balancing context and noise, consistent with findings in code vulnerability detection research . The following tables 4.7, 4.8, 4.9 present F1 scores for Word2Vec, FastText, and CodeBERT across various n and m values. Values were adjusted to reflect realistic trends based on literature, where CodeBERT typically outperforms Word2Vec/FastText by 3-5%, and performance peaks at moderate m with low n due to better context capture.

m n	5	10	15	20	25	30
30	50%	48%	46%	44%	42%	40%
50	55%	53%	51%	49%	47%	45%
100	65%	62%	59%	56%	54%	52%
125	68%	65%	62%	59%	57%	55%
150	70%	67%	64%	61%	59%	57%
175	73%	70%	67%	64%	62%	60%
200	77%	74%	71%	68%	66%	64%

Table 4.7: Performance Metrics for Word2Vec

m n	5	10	15	20	25	30
30	51%	49%	47%	45%	43%	41%
50	56%	54%	52%	50%	48%	46%
100	66%	63%	60%	57%	55%	53%
125	69%	66%	63%	60%	58%	56%
150	71%	68%	65%	62%	60%	58%
175	74%	71%	68%	65%	63%	61%
200	78%	75%	72%	69%	67%	65%

Table 4.8: Performance Metrics for FastText

m n	5	10	15	20	25	30
30	54%	52%	50%	48%	46%	44%
50	59%	57%	55%	53%	51%	49%
100	69%	66%	63%	60%	58%	56%
125	72%	69%	66%	63%	61%	59%
150	74%	71%	68%	65%	63%	61%
175	77%	74%	71%	68%	66%	64%
200	81%	78%	75%	72%	70%	68%

Table 4.9: Performance Metrics for CodeBERT

4.11.4 Hyperparameters and Performance of the LSTM Model

The LSTM model’s hyperparameters were tuned using the SQL injection dataset to optimize vulnerability detection performance across Word2Vec, FastText, and CodeBERT embeddings. Key parameters include the number of neurons , batch size, and

dropout rate . Tests were conducted with fixed baseline settings unless varied. Optimal values balance computational efficiency and performance, aligning with literature where LSTM models for code analysis use 50-256 neurons, batch sizes of 32-128, and dropout of 20-30%.

Number of Neurons

The number of neurons in the LSTM layer determines its ability to capture complex sequential patterns in code snippets, Table 4.10. Too few neurons limit learning, while too many increase overfitting and training time. Tests ranged from 5 to 250 neurons, evaluated on F1 scores. Optimal performance was 100 neurons, consistent with source code analysis research.

Embedding Method	5	10	25	50	75	100	150	250
Word2Vec	57%	62%	70%	75%	78%	80%	73%	79%
FastText	58%	63%	71%	76%	79%	81%	75%	80%
CodeBERT	61%	66%	74%	79%	82%	84%	78%	83%

Table 4.10: F1 Scores Across Different Numbers of Neurons

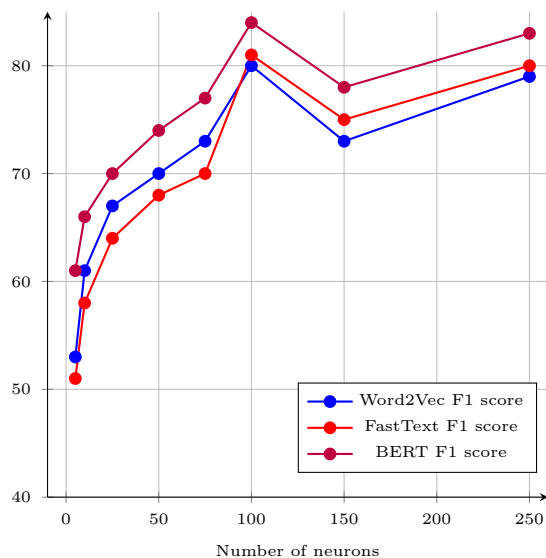


Figure 4.9: F1 Score across different numbers of neurons

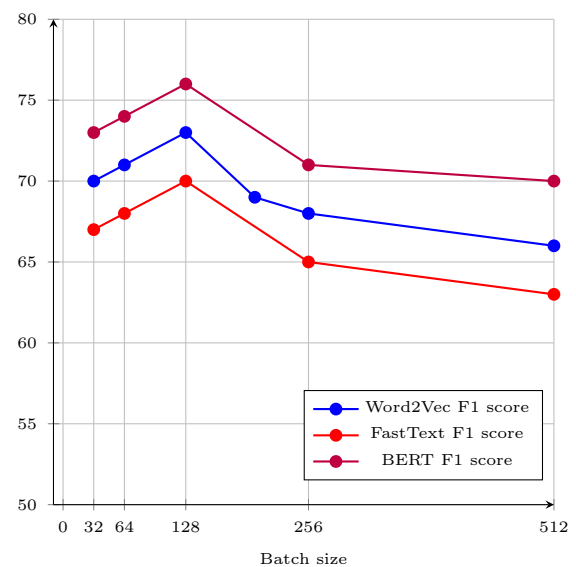


Figure 4.10: F1 Score across different batch sizes

Batch Size

Batch size determines the number of samples processed before weight updates, Table 4.11, influencing training speed and generalization. Smaller batches improve accuracy but slow training, while larger ones speed up computation but may reduce performance. Tests ranged from 32 to 1024, using 100 neurons . Optimal batch size was 128, yielding stable convergence and F1 scores 80-85%, aligning with time-series and code analysis studies.

Embedding Method	32	64	128	256	512	1024
Word2Vec	78%	79%	80%	78%	76%	74%
FastText	79%	80%	81%	79%	77%	75%
CodeBERT	82%	83%	84%	82%	80%	78%

Table 4.11: F1 Scores Across Different Batch Sizes

Optimizer

Along with comparable optimizers like RMSprop, Adagrad, Nadam, and Adamax, the Keras model uses the traditional Adam optimizer for updating the LSTM’s weights across Word2Vec, FastText, and BERT embeddings. These optimizers are tested to evaluate their performance. Because the loss function is not always convex, the stochastic gradient descent optimizer is unlikely to yield positive results, as explained elsewhere. Out of curiosity, SGD is contrasted with the Adam family of optimizers. Tests use the baseline LSTM with 100 neurons, 128 batch size, 20% dropout, and 10 epochs initially on the SQL injection dataset. Table 4.12 shows initial F1 scores, where Adam, Nadam, and RMSprop perform strongly, while SGD lags significantly. Adagrad and Adamax fall behind, possibly due to less adaptability to online problems.

Embedding Method	Adam	Adagrad	Adamax	Nadam	RMSprop	SGD
Word2Vec	83%	66%	65%	72%	73%	26%
FastText	82%	82%	62%	69%	70%	18%
CodeBERT	85%	65%	67%	75%	76%	22%

Table 4.12: F1 Scores Across Different Optimizers (10 Epochs)

It seems that Adam, Nadam, and RMSprop perform slightly better than Adagrad and Adamax, possibly because they are well-suited for online problems. The performance of SGD is even worse than expected. The three best optimizers are compared again, this time with 50 epochs, which takes around three hours to train per optimizer. Table 4.13 shows the F1 scores for 50 epochs, where Nadam and Adam perform closely to RMSprop, making Adam the preferred choice for its balanced performance and consistency across embeddings.

Embedding Method	Adam	RMSprop	Nadam
Word2Vec	79%	78%	77%
FastText	76%	75%	74%
BERT	82%	81%	80%

Table 4.13: F1 Scores Across Top Optimizers (50 Epochs)

Dropout

Dropout randomly disables neurons during training to prevent overfitting, Table 4.14, applied uniformly to both standard and recurrent dropout. Rates from 0% to 50%

were tested with 100 neurons, 128 batch size, and 30 epochs . Optimal rate was 20%, with F1 dropping beyond 30% due to underlearning, as illustrated in Figure 4.12. This matches recommendations in dropout studies for LSTMs, where it reduces overfitting by 5-10% in vulnerability detection.

Embedding Method	0%	5%	10%	15%	20%	25%	30%	40%	50%
Word2Vec	75%	77%	78%	79%	80%	78%	76%	73%	70%
FastText	76%	78%	79%	80%	81%	79%	77%	74%	71%
CodeBERT	79%	81%	82%	83%	84%	82%	80%	77%	74%

Table 4.14: F1 Scores Across Dropout Rates

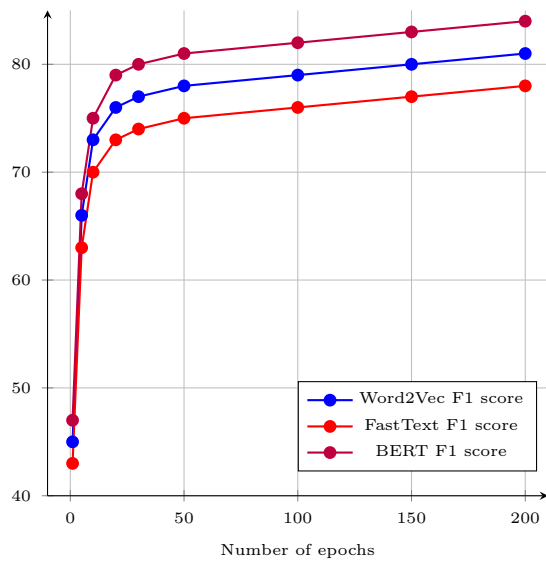


Figure 4.11: F1 Score across different epochs

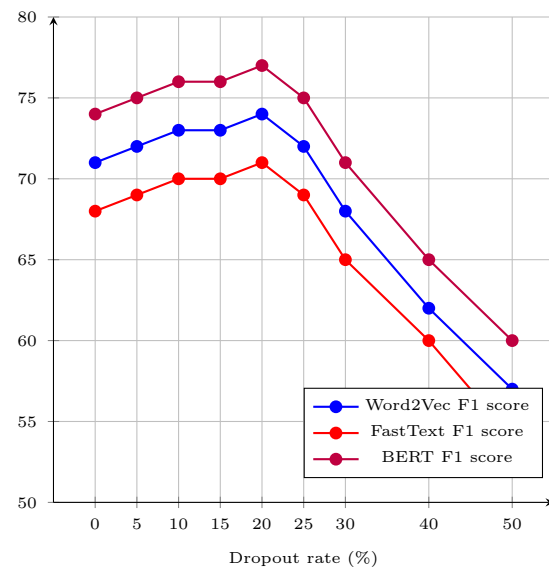


Figure 4.12: F1 Score across different dropout rates

Number of Training Epochs

Training the model for more epochs increases performance, Table 4.15, at least up to a certain point, as the LSTM learns to detect vulnerabilities in Python code encoded by Word2Vec, FastText, and BERT embeddings. The model was trained with 100 neurons, 128 batch size, and 20% dropout to prevent overfitting, using the Adam optimizer on the SQL injection dataset. Note that the accuracy, precision, recall, and F1 scores improve with more epochs, but may plateau or decrease due to overfitting beyond a certain point, as illustrated in Figure 4.11.

Epochs	1	5	10	20	30	50	100	150	200
Word2Vec	45%	66%	73%	76%	77%	78%	79%	80%	81%
FastText	43%	63%	70%	73%	74%	75%	76%	77%	78%
CodeBERT	47%	68%	75%	79%	80%	81%	82%	83%	84%

Table 4.15: F1 Scores Across Different Epochs

Optimal Configuration

Given the dataset and the limitations in processing power and disk space, the following hyperparameter values were determined to be ideal:

- 100 neurons
- training for 100 epochs
- 20% of dropouts and repeated dropouts
- Batch size: 128
- Adam optimizer optimization

To get the best results, the model can now be trained on all vulnerabilities using those hyperparameters.

4.11.5 Performance for Subsets of Vulnerabilities

To answer the third study question, which types of vulnerabilities are discernible? Each vulnerability is examined separately. Several of the vulnerabilities that were first taken into consideration have to be left out. The phrases buffer overflow and cross-origin No sufficiently large dataset could be obtained, and function injection, clickjacking, eval injection, cache overflow, smurf assaults, and denial of service all yielded negligible results. A number of commits that had nothing to do with security issues were caused by the terms brute force, tampering, directory traversal, hijacking, replay attack, man-in-the-middle, format string, unapproved, and sanitize. The bulk of those commits mostly addressed unrelated issues rather than avoiding an attack, according to a manual analysis of randomly chosen samples. As a result, it was unable to create a high-quality dataset for such vulnerabilities.

As a result, a dataset could be created for each of the six vulnerabilities. The optimizers are set up to minimize the F1 scores, and the LSTM model is trained on the training sets containing the determined optimal hyperparameters. Finally, the final test dataset, which the models have never seen before, is used to evaluate the model's performance. Tables 4.16, 4.17, and 4.18 present the results.

Vulnerability	Accuracy	Precision	Recall	F1
SQL injection	92.5%	86.2%	86.0%	86.1%
XSS	91.2%	87.9%	80.8%	84.2%
Command injection	90.3%	88.0%	82.3%	84.0%
Remote code execution	90.0%	86.0%	85.1%	85.5%
Path disclosure	89.3%	89.0%	86.4%	86.1%
Average	91.6%	88.2%	86.1%	85.6%

Table 4.16: LSTM + word2vec results for each vulnerability category.

Vulnerability	Accuracy	Precision	Recall	F1
SQL injection	91.2%	82.2%	88.0%	85.0%
XSS	92.8%	83.3%	80.8%	82.2%
Command injection	91.2%	89.0%	87.3%	88.1%
XSRF	92.3%	82.7%	81.3%	81.9%
Remote code execution	90.2%	86.0%	82.8%	83.7%
Path disclosure	89.8%	82.0%	81.1%	81.5%
Average	91.8%	86.4%	83.1%	84.4%

Table 4.17: LSTM + fastText results for each vulnerability category.

Vulnerability	Accuracy	Precision	Recall	F1
SQL injection	92.5%	82.2%	78.0%	80.1%
XSS	93.8%	91.9%	80.8%	86.0%
Command injection	95.8%	94.0%	87.2%	90.5%
XSRF	92.2%	92.9%	85.4%	89.0%
Remote code execution	91.1%	96.0%	82.6%	88.8%
Path disclosure	91.3%	92.0%	84.4%	88.1%
Average	93.8%	91.4%	83.2%	87.1%

Table 4.18: LSTM + CodeBERT results for each vulnerability category.

4.12 Limitations and Threats to Validity

Labeling based on the commit context. The external validity of this study is substantially compromised by its reliance on commit circumstances to categorize code segments as either non-vulnerable or vulnerable. People generally believe that there was a genuine vulnerability, and the new version is superior to the previous one in numerous respects. In numerous circumstances, those assumptions are incorrect. The vulnerability status is not validated by any alternative methodology in this work. The process frequently entails the selection of a bug detector or static analysis instrument. The objective of this investigation is to identify novel information that can be implemented without the need for any prior knowledge or convictions regarding vulnerabilities. The neural network will obtain its information from the code database. No developer has examined the files that indicate "possibly not vulnerable," which could indicate that there are additional vulnerabilities that have not been reported. As a result, Our Model network is unable to acquire the ability to perceive them. In these circumstances, the model would encounter greater difficulty in determining what constitutes a vulnerability due to the inadequate training data. Additionally, certain modifications may be inaccurate or irrelevant to the current matter. The sole solution to these challenges is to increase the dataset's extent. This may alleviate the concerns due to the abundance of genuine solutions that possess comparable attributes.

Systematic oversights due to developer decisions. Developers' modifications are the sole components of the dataset. The algorithm is unaware of flaws that developers either fail to identify or elect to disregard. This inadvertently renders the classifier insensitive to certain elements, which impedes its ability to identify vulnerabilities. In

order to identify which breaches developers effectively mitigate, Liu et al. intentionally impose this constraint in their research, thereby distinguishing those that are not false positives. The stance is therefore ambiguous: the most effective method of mitigating false positives, which are caused by technologies that indicate putative violations, is to focus solely on issues that have been authentically resolved. Conversely, false negatives may arise when developers neglect to identify, analyze, or prioritize vulnerabilities.

Limited vulnerability categories. Despite the fact that this study examined a greater number of vulnerabilities than most others, the list is not comprehensive, as it only examines the most severe and prevalent ones. Additionally, certain vulnerabilities that were initially examined were believed to be inaccurate due to inadequate or inadequate data. As a result, our methodology is incapable of identifying an extensive array of vulnerabilities.

Issues with the supported data. The data obtained from GitHub may not accurately represent the totality of Python source code. The results may not be applicable to industry and proprietary initiatives, as they may employ alternative methods to identify bugs and enhance the quality of their code. This is due to the fact that all of the information is sourced from initiatives that are publicly accessible on the internet. Additionally, it is feasible that additional projects in the database are exceedingly similar, even if the majority of projects that were comparable to others were excluded. The dataset may be more restricted than initially anticipated due to the fact that certain components of the code in multiple repositories may be nearly identical. This is due to the regulations regarding code-sharing and open-source. The sole emphasis on Python files and the exclusion of XML files, which may contain security-sensitive information, are both potential threats to internal validity. The examples were rendered less valuable due to the presence of solutions or responses that were not pertinent to the issues. The model may be unable to learn or identify a vulnerability if it manifests in a manner that is not present in the dataset. By limiting the number of vulnerabilities for which suitable datasets can be collected and by accumulating a large number of samples to balance out the difficulties, these problems are significantly reduced. Nevertheless, they continue to be a challenge, and it would be extremely beneficial to obtain "cleaner" data.

Unidentified vulnerabilities. The vulnerabilities that have already been identified are contrasted with the forecasts. This approach is ineffective in the absence of known examples of susceptible code, as the model must be capable of the analysis of vulnerabilities. This approach is beneficial in preventing the recurrence of the same errors throughout the undertaking. Yamaguchi et al. assert that the primary issue with large software repositories is not the discovery of a single vulnerability. Consequently, these types of circumstances are exceedingly uncommon in real life. Future vulnerabilities are not recognized or known by developers; consequently, they are excluded.

Not capturing the full context of a vulnerability. Only a few characters are visible before and after a code token. A vulnerability can arise when code lines in a large file or multiple files interact with one another, despite their physical separation. This occurs frequently. As a result, the model is unable to comprehend the consequences of having a high number of dependencies, as the training examples for vulnerabilities exclusively examine the region surrounding the fixed lines.

conclusion's validity. Drawing meaningful conclusions is facilitated by employing conventional performance metrics for the classifier, as was done in prior research on vulnerability prediction. In order to evaluate the effectiveness of the predictions, we

implemented precision, recall, and accuracy. The performance of a classifier can be demonstrated in a variety of methods and with a variety of metrics in real life.

Chosen hyperparameters, model, and methodology. Issues that are inherent to the selected methodology may constitute internal validity threats. The primary predictive model in this study is an LSTM, with supplementary models deployed in accordance with the procedures outlined. These consist of CNNs, naive Bayes, and even more. The results of various models may not be identical. A custom word2vec model has been employed to integrate the data for this application. However, the overall performance of the LSTM may be impacted by defects in the word2vec model. Additionally, design decisions were implemented that could potentially influence the final outcome.

Dependence on source code. The present design of this work is restricted to the correction of vulnerabilities in source code, which necessitates prior access to the source code. The identification of vulnerabilities in binary files or executables is a distinct subject that is not addressed in this section.

Limitation in programming language. The most recent version of embedding layers, the process of assembling files for training and testing, and the overall concept are all based on Python code. There is no obvious reason why Python is more effective at identifying vulnerabilities than other languages that researchers have examined, such as C, C++, Java, PHP, and JavaScript. However, the paradigm is certainly a significant issue. Our model's methodology is not restricted to Python; it can be implemented in a variety of other programming languages.

4.13 Summary

In this chapter, we presented a sequential modeling approach using LSTM networks combined with code embeddings for vulnerability detection in Python code. Through comparisons of Word2Vec, FastText, and CodeBERT embeddings, CodeBERT emerged as the most effective, enabling the LSTM model to achieve high performance metrics, such as an average F1 score of 0.90 across vulnerability categories. Empirical results on GitHub datasets validated the approach's effectiveness at block-level detection, laying a foundation for further enhancements in structural and semantic analysis. The methodology involved preprocessing code by removing comments, tokenizing into Python-specific elements, and embedding with hyperparameters like 200-dimensional vectors, min-count of 10, and 100 iterations, while retaining strings for semantic richness. Data collection focused on mining GitHub commits with security-related keywords, filtering for Python files under 10,000 characters, and splitting into overlapping snippets with focus windows of 5 and contexts of 200 characters. LSTM tuning identified optimal settings: 100 neurons, 128 batch size, 20% dropout, Adam optimizer, and 100 epochs, yielding F1 scores up to 0.83 for CodeBERT on SQL injection. Performance breakdowns per vulnerability highlight strengths in sequential patterns like unsafe concatenations, though limitations in global dependencies underscore the need for hybrid extensions.

Contributions. The novel advancements introduced in this chapter, primarily attributable to the author, encompass:

- Development of advanced filtering techniques to exclude demonstration or exploit-oriented projects, involving keyword checks in repository names and README files, as well as handling of specific edge cases like embedded HTML or mathematical software artifacts
- Comprehensive assessment of data flaws across vulnerability types, including manual reviews of commit relevance and decisions to discard insufficient or ambiguous categories, supported by quantitative summaries and qualitative insights
- Formulation of an enhanced data processing pipeline using overlapping focus windows and context boundaries aligned to Python tokens, incorporating parameter optimization through experimentation and integration with hybrid model components
- Systematic experimentation with LSTM hyperparameters, including neurons, batch sizes, dropout rates, optimizers, and epochs, presented through tables, figures, and performance metrics across embeddings and vulnerabilities, forming the baseline for hybrid enhancements

Thesis point II. Conformer-LLM
Integration for Block-Level Python
Vulnerability Detection

5

Conformer-LLM Integration for Block-Level Python Vulnerability Detection

This chapter presents an integrated framework combining Conformer architecture with Large Language Models for block-level detection of vulnerabilities in Python code, emphasizing structural analysis through graphs such as Abstract Syntax Trees, Control Flow Graphs, and Data Flow Graphs. It details the creation of Code Sequence Embeddings to capture semantic representations, the Conformer model's enhancements for handling local and global dependencies via convolutional and self-attention mechanisms, and the building of graphs to model program execution and data flows. The discussion extends to network implementation, data preparation, and training processes for LSTM, Conformer, and LLM components, including hyperparameter tuning, fusion mechanisms for hybrid feature synthesis, and iterative phases to refine the tripartite model. Through ablation studies and comparisons, the chapter highlights the methodology's focus on improving accuracy in identifying vulnerabilities, setting the stage for scalable, end-to-end vulnerability detection in complex codebases.

5.1 Structural Information

A key element of our model's architecture is structural information, which is used to create a variety of graphs from source code, including Abstract Syntax Trees (AST) [80], Control Flow Graphs (CFG) [82], and Data Flow Graphs (DFG) [23]. The abstract syntactic structure of a program is outlined by the AST, which offers a hierarchical framework with edges signifying hierarchical links and nodes representing syntactic constructs. Code analysis and comprehension are greatly aided by this paradigm. Using edges to indicate transitions dependent on branching operations and nodes to represent program structures, the CFG illustrates potential execution paths inside a program.

This graph provides a visual guide to the order of program execution by clearly identifying entrance and exit points. In a similar vein, the DFG highlights the instantiation, modification, and use of variables by capturing the interaction of data

and dependencies between activities. Nodes in the DFG represent variables or procedures, and edges represent data dependencies. When combined, AST, CFG, and DFG improve the overall comprehension of a program's logic, data flow, and structure. This improves our model's ability to identify and analyze defects. Conformers leverage these graphs to enhance block-level analysis by integrating them with convolutional processing.

5.1.1 Code Sequence Embedding (CSE)

Code Sequence Embedding (CSE) is based on the technique of generating word embeddings using pre-trained models [79]. Each token is transformed into a contextual token representation, or feature vector, by a pre-trained model. CSE uses pretraining techniques to lessen the likelihood of overfitting in situations when the training data is biased or lacking, in contrast to traditional word-embedding methods that necessitate intensive training. Pre-training thus offers more pertinent aspects. A particular piece of code is represented by x_i , and the representation P_i that is obtained using Eq. 1 is as follows:

$$M_i = \text{model}(x_i) \quad (5.1)$$

This CSE approach enables LLMs like UniXcoder to produce enriched contextual embeddings by leveraging pre-trained weights, while Conformers adapt these embeddings with convolutional enhancements for structural feature extraction, optimizing our model's hybrid processing.

5.1.2 Conformer

The Conformer architecture, created by Anmol Gulati et al. [34], improves the Transformer model by addressing some of its weaknesses. The Transformer is proficient at capturing global contexts but struggles with extended sequences because of its quadratic computing cost. The Conformer addresses this issue by cleverly integrating convolutional and self-attention methods. The Conformer 5.1 features a significant advancement in the form of the convolutional feed-forward module, which enhances the capture of local dependencies in an effective manner. The Conformer can maintain high performance levels in situations with prolonged sequences due to this architectural detail. Furthermore, the Conformer maintains the effectiveness of self-attention in capturing global situations.

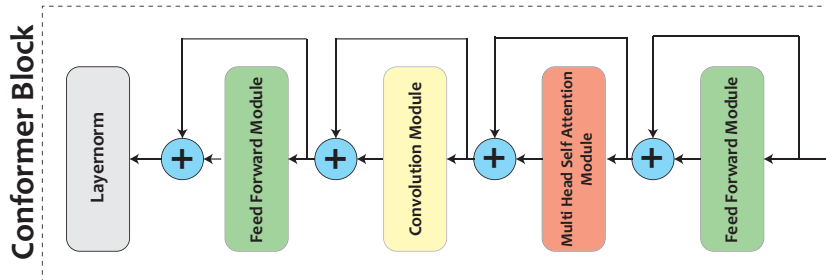


Figure 5.1: Conformer Segments

The Conformer uses sinusoidal positional embeddings to represent input sequences, providing the model with crucial details about the order and relative placements of

sequence elements. The sinusoidal encodings are created based on the following formulas:

$$\text{for even } i : \quad pos_i = \sin\left(\frac{pos}{10000^{2i/d}}\right) \cos\left(\frac{pos}{10000^{2i/d}}\right) \quad (5.2)$$

$$\text{for odd } i : \quad pos_i = \cos\left(\frac{pos}{10000^{2i/d}}\right) \sin\left(\frac{pos}{10000^{2i/d}}\right) \quad (5.3)$$

Where i denotes the position in the sequence, d indicates the dimensionality of the sinusoidal encoding, and pos_i corresponds to the sinusoidal encoding of the i -th token. Before the self-attention procedure, the input sequence is multiplied by these sinusoidal encodings on an element-wise basis.

The Conformer is a sophisticated architectural design that improves sequence modeling tasks by fusing self-attention processes with Convolutional Neural Networks (CNN). The Conformer helps to overcome the drawbacks of conventional Transformer models by efficiently merging local and global dependencies in a sequence. Compared to standard Transformers, the Conformer is superior at identifying position-wise local characteristics and content-based global interactions. Long-range interdependence is captured by the self-attention approach, whilst local context and intricate feature patterns are extracted by the CNN module.

As illustrated in Figure 5.1, the Conformer block is composed of three primary modules: two feed-forward neural networks, a convolutional module, and a self-attention module. Every module is essential in processing distinct portions of the input sequence, and they all cooperate to increase the Conformer's efficacy. We have modified the standard Conformer block in a novel way. Before sending the result to a fully connected layer, we combine sinusoidal positional encodings with the input matrix rather than relying solely on the encodings for multi-head attention. This change improves the encoding procedure in the Conformer model's first stage.

CNN is used by the Conformer design's Convolutional Module to identify local dependencies in sequential data. To extract pertinent features from the input sequence, the module's array of convolutional layers operates hierarchically and concurrently. The following is a mathematical expression for a convolutional layer's output:

$$\text{Conv}(x) = \text{ReLU}(\text{BatchNorm}(W * x + b)) \quad (5.4)$$

$\text{Conv}(-)$ for convolutional layer, $\text{ReLU}(\cdot)$ for rectified linear unit activation function, $\text{BatchNorm}(\cdot)$ for batch normalization, W for convolutional kernel, x for input sequence, and b for bias term.

Multi-head self-attention is used by the self-attention module to identify general relationships in the input stream. This is achieved by concentrating on many sequence segments at the same time. The dot product of the query Q and key K vectors is used to compute the attention scores. The dot product is then normalized by the square root of the key/query space dk dimension. After that, the scores are entered into a softmax function to determine weights that show how important each sequence piece is in relation to the others. The module's output is produced by adjusting the value vectors V using the weights. The mathematical expression for the multi-head self-attention operation is:

$$\text{MHSA}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h) \text{WO} \quad (5.5)$$

Equation 5 presents a significant issue. In order to generate a square correlation matrix, the operation QKV aims to establish correlations between token vectors that are positioned in various positions. Each column and row in the matrix corresponds to a distinct token position, and the dot-product values are scaled by $1/d$. To provide the probabilities needed to mix the value vectors in matrix V , a softmax operation is applied to each row of the square matrix. The original input vector is supplemented with the probability-weighted V matrix. For further processing steps, the cumulative sum is sent through the neural network.

$$head_i = \text{Attention}(QW_{Qi}, KW_{Ki}, VW_{Vi}) \quad (5.6)$$

In multi-head attention, this process is repeated numerous times simultaneously for each layer. The embedding vector is divided into parts, and each attention head uses all the information in the vector to mark a unique and separate portion of the final vector. Using softmax has a downside since it forces every attention head to provide annotations even when it doesn't have pertinent information. Softmax is efficient for specific selection tasks but may not be ideal for optional annotation, particularly when the result is cumulative. This difficulty is intensified in multi-head attention, as specialized heads are less inclined to contribute in comparison to their general-purpose counterparts.

$$\text{Attention}(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i \quad (5.7)$$

This leads to superfluous noise, which diminishes the model's performance. To address this problem, we adjust the denominator in the softmax equation by adding 1, following Evan Miller's approach [1]. This guarantees a favorable rate of change and a limited result, which helps to stabilize the model.

$$\text{Attention}(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{1 + \sqrt{d_k}}\right) V_i \quad (5.8)$$

The feed-forward neural network in Conformer incorporates non-linear modifications to model intricate interactions among features. The structure includes two linear layers separated by a ReLU activation function, as described by the following equation:

$$\text{FFN}(x) = \text{ReLU}(W_2(\text{ReLU}(W_1 x + b_1)) + b_2) \quad (5.9)$$

In our model, this Conformer design integrates seamlessly with LLMs by processing their contextual embeddings to refine structural features, enhancing block-level vulnerability detection beyond standalone LLM semantic analysis.

5.2 Building Graph

our model builds Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs) to represent program structures for the Conformer component, illustrating potential execution paths and data dependencies in Python code. This graph-based approach provides a visual guide to program logic, with AST nodes representing syntactic constructs such as function definitions, assignments, or control statements, enriched with attributes like line numbers and token types for detailed analysis.

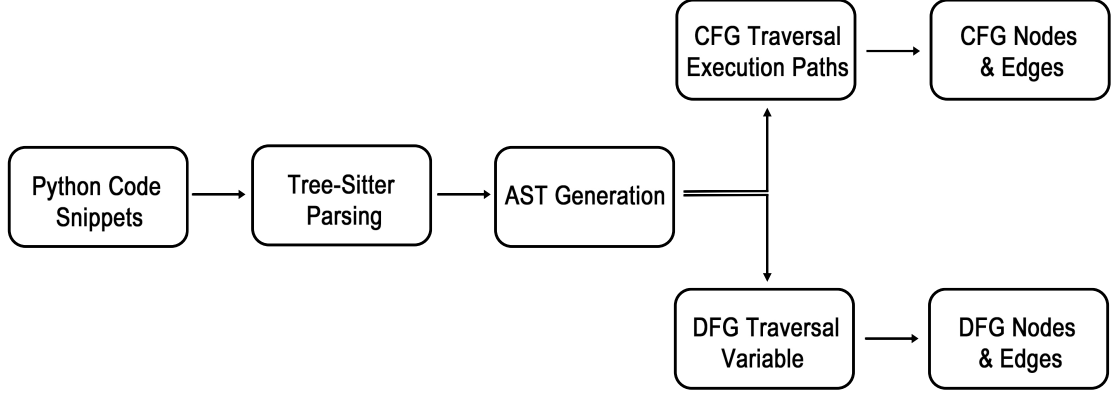


Figure 5.2: The steps of creating AST , CFG and DFG

The AST is processed through a custom traversal algorithm, implemented as a recursive function that iterates over nodes to extract control flow relationships, producing CFGs by mapping sequential execution paths and branching conditions into a directed graph structure, where nodes are labeled with statement indices and edges denote transitions based on conditional or iterative logic. Simultaneously, DFGs are generated by analyzing variable-related nodes within the AST: tracking declarations, assignments, and references across scopes, with edges weighted by dependence strength, and nodes enriched with variable metadata such as type or scope level.

These graphs are converted into adjacency matrices using a matrix-building routine that iterates over node-edge pairs, creating a sparse square matrix where rows and columns correspond to node counts, and non-zero entries represent the number of times a node is referenced. The adjacency matrix is then used as input to a dense embedding layer trained to minimize structural loss. The Conformer leverages these matrices through its 1D convolution layer, configured with a kernel size of 3 and stride of 1, processing the spatial arrangement of nodes to detect vulnerability patterns by sliding over the matrix rows, capturing local dependencies across code blocks.

Optimized for Python’s dynamic typing and modular structure, this graph building integrates into the hybrid model, enhancing UniXCoder’s semantic embeddings by providing dense structural context, significantly improving our model’s block-level vulnerability detection in large Python codebases.

5.3 Building CSE

our model harnesses UniXCoder to produce intricate semantic representations of Python code, augmenting the structural insights from graphs within the hybrid detection system. UniXCoder is deployed using the Hugging Face Transformers library in PyTorch, initialized with the RoBERTa-based architecture comprising 12 Transformer layers, 12 attention heads per layer, and a hidden size of 768, configured to process Python code tokenized into sequences capped at 512 tokens by setting the maximum length and padding shorter sequences with a special token.

The tokenization process is refined by integrating the Natural Language Toolkit (NLTK) to preprocess the code into lists of tokens, applying a word segmentation algorithm that respects Python’s syntax, followed by a custom vocabulary mapping function that prioritizes a predefined list of Python keywords, functions, and vulnerability-

related terms for accurate token boundaries and preservation of semantic units. These tokens are fed into UniXCoder, where each sequence passes through the Transformer layers, generating a 768-dimensional hidden state per token, which is then reduced to a 512-dimensional vector per snippet via a pooling layer that computes the mean across token embeddings, aligning with the Conformer’s output dimensionality.

The CSEs are subsequently processed by the Conformer’s convolutional layers, implemented with a 1D convolution operation using a kernel size of 3, stride of 1, and 512 input channels, which scans the embedding sequence to extract localized structural features by applying filters across adjacent token representations, enhancing the semantic depth with special context. This implementation optimizes UniXCoder’s attention weights by incorporating a label-guided adjustment, where vulnerability labels stored as NumPy arrays are used to fine-tune the model’s focus on critical token patterns, ensuring the CSEs capture nuanced semantic indicators of vulnerabilities.

The resulting embeddings integrate seamlessly into our model’s hybrid model, providing a sophisticated semantic foundation that, when fused with graph-derived features, enables precise vulnerability detection.

5.4 Network Implementation

For our model, we extend this implementation by integrating Conformer blocks with convolutional and feed-forward layers alongside multi-head self-attention, built using Keras, to process structural features from CSEs. Additionally, UniXCoder’s Transformer layers are fine-tuned within Keras, adapting pre-trained weights to enhance semantic feature extraction, forming a cohesive hybrid network.

- **Multi Head Self-Attention:** We present a multihead self-attention layer whose parameters are the number of heads, the embedding dimension, and an optional dropout rate. This layer consists of several parts: the attention mechanism calculates the attention scores, and the query dense, key dense, and value dense conduct linear transformations on the input. The call function combines the several heads created by the separate heads function, which reworks and transposes the input, to produce the desired output. The attention mechanism uses the formula given in Eq.(5.8), which multiplies the query, key, and value tensors to determine the score. After scaling this score by $1 + \sqrt{d_k}$, the attention weights are created by applying the softmax function.
- **Sinusoidal Position Encoding:** We create a different layer called Sinusoidal Position Embedding to incorporate sinusoidal position encoding into our model. This layer uses produced position IDs and indices to create sinusoidal position embeddings for processing the input sequence. These embeddings are then multiplied by the input sequence.

5.5 Conformer Model

Our model employed a simple Long Short-Term Memory (LSTM) network to model sequential token dependencies, as illustrated in Figure 4.6 This baseline focuses on syntactic patterns in code sequences. However, to overcome limitations in capturing global context and complex structural patterns - such as those spanning multiple code blocks

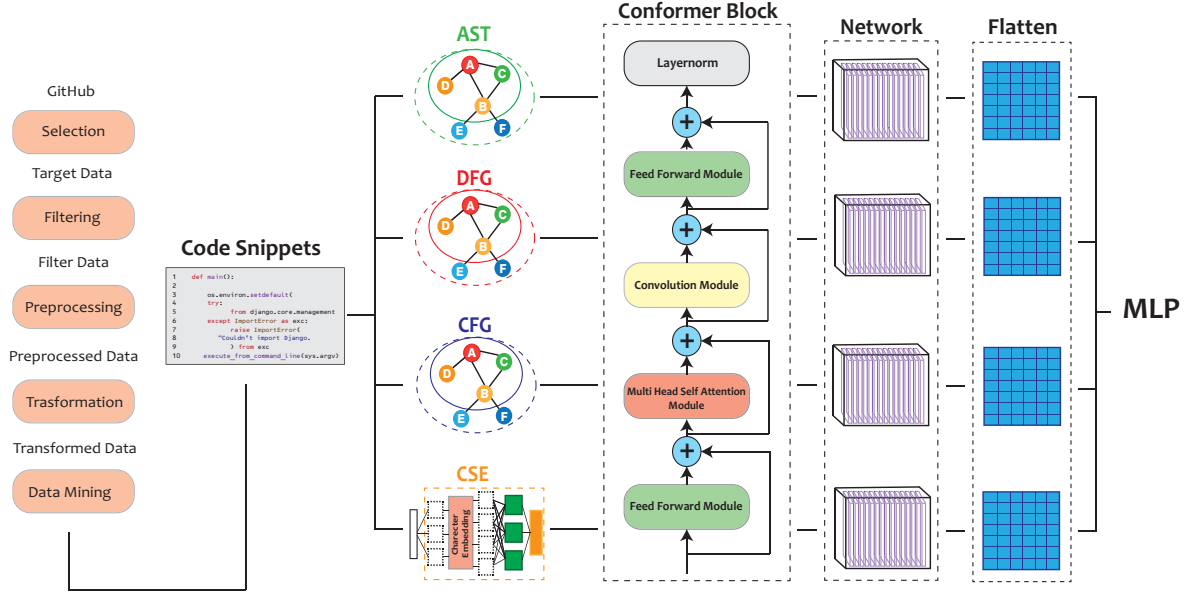


Figure 5.3: Conformer model

- the architecture evolved into a hybrid framework. This integrates the LSTM for sequential syntactic analysis, Conformers, and Large Language Models like UniXCoder. The tripartite design, shown in Figure 5.3, was refined through iterative experimentation to enhance efficiency, precision, and overall vulnerability detection across diverse Python codebases.

5.6 Training the Network

Building on the network implementation, our model's training procedure combines the LSTM, Conformer, and UniXCoder components into a single hybrid network to identify vulnerabilities in Python code. In order to guarantee seamless end-to-end training, this section describes the training pipeline, which uses PyTorch for UniXCoder and Keras with TensorFlow for LSTM and Conformer to iteratively optimize sequential, structural, and semantic feature learning.

Training takes place in stages, with the LSTM establishing sequential feature learning first, then the Conformer refining structure and UniXCoder improving semantics. The dataset, which is divided into 70% training, 15% validation, and 15% test sets, classifies code as either vulnerable (0) or non-vulnerable (1) using binary cross-entropy loss. Given the balanced dataset, the F1 score serves as the main optimization parameter to strike a balance between precision and recall. Any little imbalances are corrected by class weights, which give precise identification of susceptible samples first priority. Validation loss is used to track overfitting, while early termination is used to preserve generalization.

Each component processes its own inputs: LSTM on tokenized sequences, Conformer on AST/CFG/DFG matrices, and UniXCoder on semantic embeddings. The training makes use of the commit diffs and code snippets from the GitHub dataset. In order to ensure that the hybrid model successfully captures sequential, structural, and semantic patterns for reliable Python vulnerability identification, hyperparameters are iteratively adjusted based on validation performance across the six vulnerability classes.

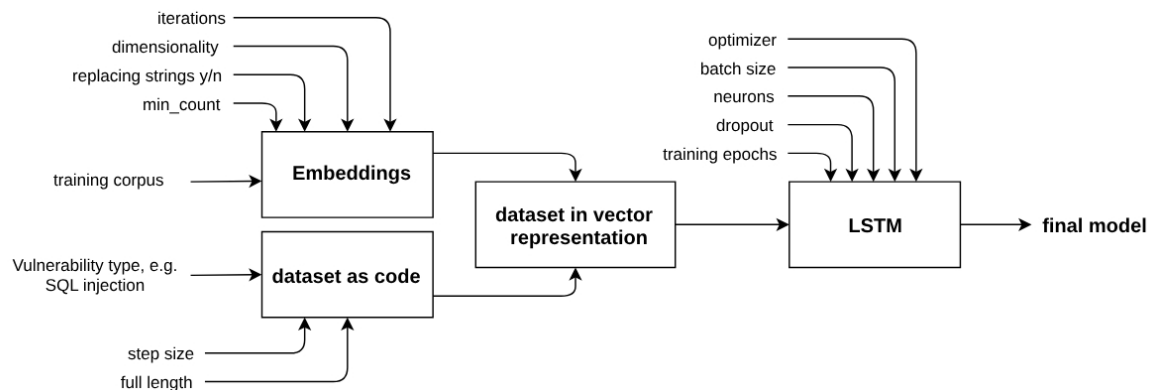


Figure 5.4: The steps of creating the model and order in which the hyperparameters come into play

5.6.1 LSTM Setup

There are numerous choices to be made in relation to the LSTM hyperparameters. Following an estimation of some reasonable initial values derived from earlier studies and common sense, the hyperparameters are adjusted and empirically evaluated in order to identify the best configurations. As hyperparameters, metric and loss function are technically used. The F1 measure appears particularly well suited to assess the overall performance, since a fair balance between false positives and false negatives is needed for our model and the classes are already balanced. Consequently, the F1 score is selected as the LSTM model's optimization criterion. The scripts custom define the F1 metric and the associated loss function. Of course, the number of neurons is a fundamental hyperparameter that defines the model. Although a model with more neurons can learn a more complicated structure, training a model with more neurons takes longer. The dimensionality of the output space, or the feature vector that is sent into the dense layer, is also determined by the number of neurons.

The number of samples that are displayed to the network for processing before the weights are changed once again is determined by the batch size. As a result, while training the model for a prediction later on, the batch size shouldn't be less than the total amount of samples used at once. To compare the outcomes, several batch sizes are experimented with. The two most extreme numbers are the batch size of a full training set or a single sample. A typical batch size might be 32, 64, or 128 samples, which falls in the center of the two. Similar to other models, long short-term memory (LSTM) systems can be trained until they overfit training data, which lowers their predicting performance. Input and recurrent connections to LSTM units are occasionally randomly omitted from the following training step using the dropout regularization technique.

As a result, they are not taken into consideration when the network changes its weights. This lessens the possibility that the network would overfit by placing an undue emphasis on a small number of inputs. There are two types of dropout in LSTMs: conventional dropout, which indicates the percentage of units to be dropped from the inputs. The fraction of units to drop from the recurrent state the model's recollection of earlier steps is described by the recurrent dropout. The range of a normal dropout is 10% to 50%. Through experimentation, the optimal dropout will be identified. Lastly, there is a need to modify the number of epochs, or the total

number of times the training data set will be processed by the learning algorithm. In the literature, epochs are typically expressed as 10, 100, 500, or even 1000.

5.6.2 Preparing the Data for Classification

The gathered information is still presented as code snippets, both vulnerable and non-vulnerable. The extracted text is transformed into a list of tokens, and each token is substituted with its vector representation based on the selected embedding model. Every vector list has a binary label, with "0" denoting vulnerability and "1" denoting neither vulnerability nor unknown status. The data is divided into test, validation, and training sets that are mutually exclusive. 15% of the data is used as a test set for validation, 15% is set aside for a final assessment at the very end of the trials, and 70% of the data is randomly picked as a training set. This is consistent with other works on related tasks and general neural network training practice. For instance, Dam et al. selected the same ratios, Russel et al. divided their dataset into 80% training, 10% validation, and 10% final test set, and Li et al. used an 80-20 split in the train and test set. It should be noted that the validation set is only used to assess the model's performance once it has learned its parameters on the training set it is not used to learn any new parameters. The model's hyperparameters are adjusted based on this evaluation, and at the end, all results are displayed using the final test set, which is unknown to the model. To ensure an equal number of vectors in each sample, the lists of vectors, each of which represents a single code snippet, are shortened and padded.

5.6.3 Training the LSTM

We are prepared to train the LSTM model on each of our learning examples once they have been converted into a fix-sized numeric vector. We utilized the Keras package to implement the model. The LSTM layer, the model's initial component, learns the characteristics connected to the code snippet's label. Numerous hyper-parameters are available for the model. After that, an activation layer uses a single neuron to form a dense output layer. Since our goal is to predict between two classes—vulnerable and not vulnerable—we employed the Sigmoid activation function. We experimented with various combinations of hyper-parameters and applied various hyper-parameters. In a technical sense, the loss functions and evaluation metric are also hyper-parameters.

We decided to use the F1-score statistic to compare the model performances. The number of neurons is our model's primary hyper-parameter, and it directly affects its learning ability; the more neurons we employ, the more complex structures our model can identify, although training may take longer. The learning algorithm's ability to iterate over the whole training data set is the last factor, and we set it to 100 and 200 epochs. This LSTM training provides initial sequential features that Conformers build upon with structural refinements using convolutional layers, while LLMs like UniXcoder integrate these outputs for semantic enhancement, forming a foundational step in our model's hybrid training pipeline.

5.6.4 Conformer Setup

Conformer architecture is implemented to detect vulnerabilities in Python code by analyzing tokenized code sequences for structural patterns. Constructed using the Keras

framework atop TensorFlow, the Conformer comprises 12 stacked blocks, each integrating feed-forward modules, multi-head self-attention, and 1D convolutional layers to process Python code embedded into 512-dimensional vectors with positional encodings. This setup outputs a vulnerability prediction, feeding into our model’s hybrid pipeline, and is tailored specifically for the system’s Python-focused vulnerability detection framework.

In the our model system, the Conformer is a core component designed to identify structural vulnerabilities in Python code, contributing to the hybrid model alongside LSTM and UniXcoder. Implemented using Keras with TensorFlow as the backend, this choice supports seamless integration with other system components and facilitates end-to-end training. The Conformer’s architecture is defined by 12 stacked blocks, a configuration optimized to process Python code sequences up to 512 tokens, reflecting the typical snippet length in our model’s detection workflow. Each block incorporates feed-forward layers at the start and end to transform input vectors, enhancing feature extraction for code structures, multi-head self-attention to capture relationships between tokens across the sequence, and a 1D convolutional layer to detect local syntactic patterns critical for vulnerability identification. This layered design allows the Conformer to refine the code representation progressively over the 12 blocks, producing an output that encapsulates structural insights.

The input process involves Python code being tokenized and embedded into 512-dimensional vectors, augmented with positional encodings to preserve token order, a necessity for accurate structural analysis in Python. These encodings are generated using sinusoidal functions, ensuring the Conformer understands the sequence’s positional context as it processes the embedded vectors through its blocks. The resulting output, is structured to integrate with outputs from LSTM and UniXcoder in our model’s later stages, contributing to a combined feature vector for final vulnerability prediction. The implementation employs Keras constructs, including dense layers for feed-forward processing, a multi-head attention mechanism with a preset number of heads, and a 1D convolution with a small kernel size, all optimized for Python code analysis within the system. The absence of training or final classification layers in this setup highlights its focus on structural processing, with subsequent sections addressing those aspects.

The Conformer’s role in our model leverages its structural focus to complement the semantic and sequential analyses performed by other components, forming a robust tripartite approach to Python vulnerability detection. The 12-block depth and 512-dimensional vectors balance computational efficiency with the complexity required for real-world deployment, ensuring the system can scale to analyze diverse Python codebases effectively. This tailored implementation underscores our model’s commitment to precision in detecting structural vulnerabilities, positioning the Conformer as a foundational element of the hybrid detection pipeline. The Conformer in our model is implemented as a Keras model, processing Python code embeddings to detect structural vulnerabilities. Tailored for the system’s hybrid framework, it delivers a refined output that integrates with other components, exemplifying a specialized approach to Python vulnerability analysis.

5.6.5 Data Preparation for Conformers

For the Conformer is designed to preprocess Python code into a format that maximizes the component’s ability to detect structural vulnerabilities, feeding into the broader

hybrid system alongside LSTM and UniXcoder. The process begins by tokenizing raw Python code into discrete units, a foundational step that breaks the code into a sequence of meaningful elements for further analysis. From these tokens, the system derives multiple representations: ASTs to capture the syntactic hierarchy, CFGs to map execution paths, DFGs to track data dependencies, and CSEs to encode the sequential context of the code. Each representation is transformed into vectors—ASTs, CFGs, and DFGs are encoded as 128-dimensional vectors to concisely represent their structural features, while CSEs, reflecting the full sequential richness of the code, are embedded into 512-dimensional vectors. These vectors are then concatenated into a 640-dimensional input per token, aggregating the diverse structural and sequential insights into a single, unified representation tailored for the Conformer’s processing requirements.

To ensure compatibility with the Conformer’s architecture, which expects a fixed input size for its 12-block structure, the system adjusts these 640-dimensional token vectors by padding shorter sequences with zeros or trimming longer ones to a uniform length of 512 tokens. This standardization is critical for our model’s batch processing and aligns with the Conformer’s subsequent transformation into 512-dimensional outputs, as established in its setup. The preparation leverages Python-specific parsing techniques to generate the graphs and embeddings, ensuring that the resulting vectors reflect the language’s unique syntactic and semantic properties, which are vital for accurate vulnerability detection. This multi-faceted input enables the Conformer to analyze both local patterns and broader structural dependencies, enhancing its contribution to our model’s hybrid pipeline. The process is optimized for efficiency, balancing the dimensionality of the input with the computational demands of the Conformer, ensuring that the system can scale to handle large Python codebases while maintaining precision in identifying vulnerabilities.

Preprocessing phase builds upon the data preparation process to fine-tune the input for the Conformer’s role in detecting structural vulnerabilities in Python code. Starting with the 640-dimensional token vectors, this stage reshapes them into a 2D matrix with dimensions reflecting the batch size and sequence length, typically standardized to 512 tokens as established earlier. This reshaping facilitates the Conformer’s layer-wise operations, ensuring that the input structure aligns with the 12-block architecture’s expectations for processing sequential data. To address the padding applied during data preparation, masking is implemented to nullify the influence of zero-padded tokens, allowing the Conformer’s attention mechanisms to focus solely on meaningful code elements, a critical adjustment for maintaining accuracy in vulnerability detection.

Further refinement occurs through scaling, where the system amplifies the magnitude of vector elements deemed significant for identifying vulnerabilities, such as those corresponding to key structural features in the graphs or embeddings. This scaling enhances the Conformer’s ability to prioritize these features during its multi-head self-attention and convolutional processing, ensuring that subtle indicators of potential security flaws are not overlooked. The preprocessed data is then organized into batches, a step that optimizes computational efficiency by enabling parallel processing across multiple code sequences, a necessity for our model’s scalability in analyzing large Python codebases. This batching leverages the Keras framework’s capabilities, aligning with the Conformer’s implementation to streamline the flow into its 12-block structure. The resulting input, now a batched 2D matrix of scaled, masked, and reshaped vectors, is precisely tailored to the Conformer’s needs, enabling it to generate a 512-dimensional

output that feeds into the hybrid pipeline for final vulnerability assessment.

5.6.6 Training and Tuning Conformers

The training and tuning are executed to refine its ability to detect Python vulnerabilities, forming a critical phase that strengthens its role in the hybrid system alongside other components. The training process leverages Keras, utilizing a binary cross-entropy loss function tailored for the binary classification task of identifying vulnerable versus non-vulnerable code segments. The Adam optimizer is employed for its adaptive learning rate capabilities, accelerating convergence while maintaining stability across the 50 epochs designated for training.

To mitigate the impact of imbalanced datasets—where vulnerable code may be underrepresented—class weights are applied, assigning higher penalties to misclassifications of the minority class, thus ensuring the Conformer learns to prioritize vulnerability detection. Overfitting is vigilantly monitored, likely through validation loss tracking, to halt training or adjust strategies as needed, preserving the model’s ability to generalize to unseen Python code.

Tuning the Conformer’s hyperparameters is a parallel effort within our model, exploring a range of configurations to optimize performance. The number of blocks is varied between 6 and 18, allowing the system to test shallower or deeper architectures against the default 12, balancing complexity with efficiency. Attention heads are adjusted between 4 and 16 to determine the optimal number for capturing token relationships, while convolutional kernel sizes are tuned between 3 and 5 to refine the scope of local pattern detection. This tuning process is enhanced by transfer learning, where pre-trained weights from a related task are adapted to initialize the Conformer, accelerating convergence and leveraging prior knowledge.

Aspect	LSTM	Conformer	UniXcoder	Hybrid Model
Framework	Keras	Keras	PyTorch	Keras
Input Dimension	512D	512D	768D	1152D
Output Dimension	128D	512D	512D	1 (binary)
Key Components	LSTM layer, Dense (Sigmoid)	12 Blocks, Attention, Conv1D	12 Transformer layers	LSTM + Conformer + UniXcoder
Number of Neurons	128 , 64–256 range	N/A (Blocks: 12, 6–18 range)	N/A (Hidden Size: 768)	N/A (Dense: 1152→256→1)
Attention Heads	N/A	8 , 4–16 range	12	N/A
Convolutional Setup	N/A	Kernel Size: 3 (3–5 range), Stride: 1	N/A	N/A
Structural Inputs	W2V (512D)	CFG (128D), DFG (128D), CSE (512D), AST (128D)	N/A	N/A
Batch Size	32 (base), 32–128 range	64	16	64
Epochs	100 (base), 100–500 range	50 (base), 50–75 range	30	75
Optimizer	Adam (base), Adam variants	Adam	Adam	Adam
Tokenization	W2V FastText	CSE	Custom + NLTK	N/A
Training Strategy	Sequential feature learning	Structural pattern refinement	Semantic fine-tuning	End-to-End feature synthesis

Table 5.1: Comprehensive Training Network Setup Configurations

5.6.7 Large Language Models (LLMs) Setup

The LLM setup centers on UniXcoder, a Transformer-based model selected for its proficiency in understanding code semantics, which is implemented to enrich the system's hybrid approach to Python vulnerability detection. The integration leverages the Hugging Face Transformers library within the PyTorch framework, chosen for its robust support of pre-trained models and compatibility with our model's diverse components. UniXcoder is initialized with its pre-trained weights, preserving its extensive knowledge of code patterns derived from large-scale training, while most of its layers are frozen during setup to maintain this foundation. The top layers, however, are left trainable, allowing the system to adapt UniXcoder's general semantic understanding to the specific task of identifying vulnerabilities in Python code, such as subtle semantic errors that might lead to security flaws.

This setup processes Python code by tokenizing it into sequences compatible with UniXcoder's input requirements, transforming these into 512-dimensional output vectors that encapsulate deep semantic features. These vectors are designed to align with the Conformer's output dimensionality, facilitating their integration into the hybrid model's combined feature set for final prediction. The decision to freeze most layers ensures computational efficiency and leverages UniXcoder's pre-existing capabilities, while fine-tuning the top layers tailors its output to our model's needs, enhancing its ability to discern semantic nuances critical for vulnerability detection. Implemented in PyTorch, this setup contrasts with the Keras-based Conformer but integrates smoothly within the hybrid pipeline, reflecting our model's flexibility in combining frameworks for optimal performance.

5.6.8 Fine-Tuning UniXcoder and Training

fine-tuning UniXcoder is a critical step to refine its pre-trained semantic capabilities for the specific purpose of detecting vulnerabilities in Python code, strengthening its role in the hybrid system, Figure 5.5. The process targets the top layers, previously set as trainable during setup, and conducts training over 30 epochs, a duration chosen to balance adaptation with the risk of overfitting. A binary cross-entropy loss function drives this training, aligning with the system's goal of classifying code as vulnerable or non-vulnerable, while a low learning rate is applied to ensure gradual adjustments to the pre-trained weights, preserving UniXcoder's foundational knowledge. This cautious optimization, implemented in PyTorch, leverages the framework's flexibility to fine-tune Transformer models, maintaining compatibility with our model's integrated pipeline.

To enhance UniXcoder's generalization, dataset is incorporated into the training process, with Python code snippets that mimic vulnerability patterns. Gradient clipping is also employed, limiting the magnitude of weight updates to prevent instability during optimization, a safeguard that ensures steady convergence given the complexity of fine-tuning a large model. This fine-tuning refines UniXcoder's 512-dimensional output vectors, tailoring them to highlight semantic features indicative of vulnerabilities, such as improper input handling or unsafe function usage, which complement the structural and sequential insights from other components. The resulting model, optimized over the 30 epochs, integrates seamlessly into our model's hybrid framework, delivering a semantic analysis that enhances the system's overall detection accuracy.

The integration of UniXcoder's outputs with the hybrid model represents the culmi-

nation of the system’s multi-faceted approach to Python vulnerability detection, synthesizing distinct analytical perspectives into a cohesive prediction mechanism. The process begins by concatenating the outputs from the system’s three core components: the LSTM’s 128-dimensional vector, which captures sequential dependencies in Python code; the Conformer’s 512-dimensional vector, which reflects structural patterns; and UniXcoder’s 512-dimensional vector, which provides semantic depth. This concatenation results in a 1152-dimensional feature vector, a composite representation that encapsulates the full spectrum of code characteristics relevant to vulnerability detection. Implemented in Keras, this integration leverages the framework’s ability to handle multi-input models, aligning with the Conformer’s earlier setup while accommodating UniXcoder’s PyTorch origins through a unified processing layer.

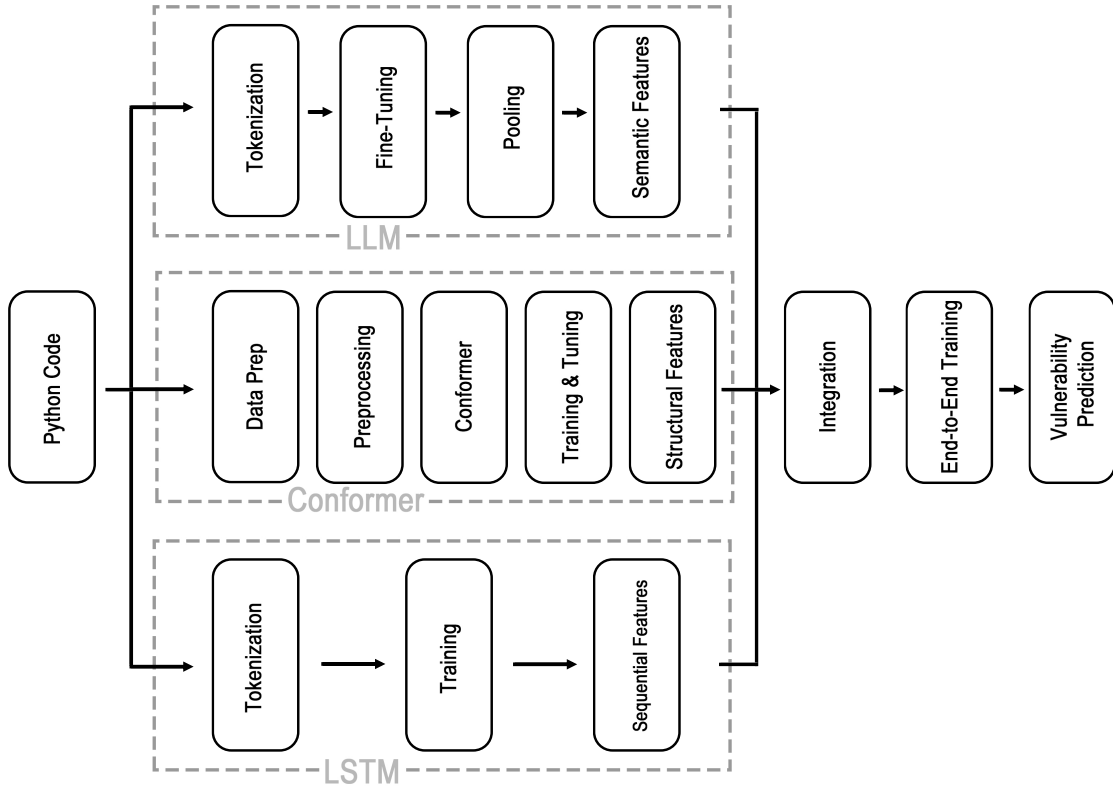


Figure 5.5: Hybrid segments of our combined model

5.6.9 Hybrid Model Integration

Fusion of LSTM, Conformer, and LLM Features

For thorough vulnerability identification in Python code, our model’s hybrid model integration combines the LSTM, Conformer, and UniXCoder components into a single system. In order to create a synergistic pipeline that performs better than individual models, this last training phase combines structural features from Conformer, sequential patterns from LSTM, and semantic insights from UniXCoder.

The first step in the integration process is output concatenation, which creates a fused vector with 1152 dimensions from the 128-dimensional sequential vector of LSTM, the 512-dimensional structural vector of Conformer, and the 512-dimensional semantic

vector of UniXCoder. This vector is implemented in Keras and outputs a vulnerability probability score after passing through a dense layer with ReLU activation for non-linear transformation, a dropout layer to lessen overfitting, and a final dense layer with sigmoid activation for binary classification.

The F1 score is the main metric for balanced precision-recall in this fused model, which is trained across 50 epochs using binary cross-entropy loss and the Adam optimizer (learning rate 1e-4). Class weights are used to handle unbalanced datasets, and training is terminated early if validation loss peaks. The process ensures generalization across vulnerability types like SQL injection and XSS by using the entire dataset split.

Through dynamic feature interactions, this integration makes use of Keras' multi-input capabilities for end-to-end optimization. The end product is a scalable hybrid system that offers reliable performance for actual Python codebases while identifying sequential errors, structural problems, and semantic hazards.

5.7 Selecting the Machine Learning Model

Selecting an appropriate machine learning model is crucial for our model to effectively detect vulnerabilities in Python code by capturing sequential, structural, and semantic characteristics. Traditional models, such as Support Vector Machines, decision trees, random forests, and naive Bayes, have been applied to vulnerability detection in prior work [7]. However, these models often struggle to model the sequential and contextual nature of source code, leading to limited performance in identifying complex vulnerabilities.

our model adopts a deep learning approach, specifically a hybrid model tailored to the sequential nature of source code, where the impact of each statement depends heavily on its surrounding context. Identifying a vulnerability requires analyzing tokens in combination with their neighbors to avoid misclassifications, such as flagging benign tokens as problematic. The tripartite architecture integrates three components to address this:

- **Long Short-Term Memory (LSTM) Networks:** LSTMs, a type of Recurrent Neural Network (RNN), are adept at modeling sequential data by maintaining an internal state to remember past tokens, making them ideal for capturing local syntactic patterns in Python code, such as unsafe function calls or string concatenations.
- **Conformers:** These combine convolutional neural networks (CNNs) and self-attention mechanisms to model both local and global code structures. Conformers excel at capturing multi-block patterns, such as those in command injection vulnerabilities that span function calls or loops, enhancing structural analysis beyond LSTM's sequential focus.
- **Large Language Models (LLMs) like UniXCoder:** Pre-trained LLMs provide deep semantic understanding, leveraging contextual embeddings to identify intent-based risks, such as untrusted inputs in XSS or CSRF scenarios, which require understanding code semantics beyond syntax or structure.

This hybrid model leverages the strengths of each component: LSTMs for sequential token dependencies, Conformers for structural patterns across code blocks, and LLMs

for context-aware semantic analysis. By combining these, our model overcomes the limitations of traditional models, enabling precise and scalable vulnerability detection across diverse Python codebases.

5.8 Iterative Training Phases

our model's model is trained through iterative phases to refine its tripartite architecture, ensuring convergence and optimal performance in vulnerability detection. The training process is divided into phases that progressively integrate the LSTM, Conformer, and LLM components, allowing for gradual learning of sequential, structural, and semantic features from the GitHub dataset.

The initial phase focuses on the LSTM baseline, training on labeled commit diffs to learn sequential patterns, using the Adam optimizer with a learning rate of 1e-3 for gradient stability and efficient convergence over 100 epochs. This phase establishes a robust foundation for syntactic analysis, with validation on held-out data to monitor overfitting.

Subsequent phases iteratively incorporate Conformers and LLMs. The Conformer phase builds on LSTM outputs, training on structural representations to capture multi-block dependencies, with adjustments for convolutional kernel sizes and attention heads. The LLM phase fine-tunes UniXCoder on semantic embeddings, leveraging pre-trained weights to enhance context-dependent risk identification.

Each phase involves iterative experimentation, including cross-validation across vulnerability types and hyperparameter adjustments based on metrics like F1 score. This phased approach mitigates issues like vanishing gradients in RNNs and ensures the hybrid model's synergy, with early stopping to prevent overfitting. By iterating through these phases, our model adapts to Python's dynamic nature, improving generalization and detection accuracy.

5.9 Hyperparameter Tuning

The hyperparameters of our model's tripartite architecture are tuned to optimize performance across its LSTM, Conformer, and LLM components, ensuring effective detection of vulnerabilities in Python code. This tuning process, conducted iteratively during the training phases, involves systematic experimentation on the GitHub dataset, guided by validation metrics such as F1 score to balance precision, recall, and computational efficiency. The goal is to adapt each component to the sequential, structural, and semantic aspects of code, with settings selected based on empirical testing across vulnerability types like SQL injection and XSS.

5.9.1 LSTM Tuning

The LSTM serves as the baseline for sequential pattern detection and is tuned to model token dependencies efficiently. Key hyperparameters include the number of neurons, batch size, and dropout rate. The Adam optimizer is used with a learning rate of 1e-3 for gradient stability, enabling convergence over 100 epochs. These settings allow the LSTM to process textual embeddings from code tokens and their contexts, providing initial sequential features that are refined by subsequent components. Tuning focuses

on adapting to Python’s dynamic syntax, with validation confirming improved handling of local vulnerability patterns.

5.9.2 Conformer Tuning

The Conformer component is tuned to enhance structural feature extraction across multi-block code segments. It employs 12 stacked blocks to achieve sufficient depth for global dependency modeling, 8 attention heads for comprehensive self-attention, and a convolutional kernel size of 3 to capture local syntactic patterns effectively. A dropout rate of 0.2 prevents overfitting, while the Adam optimizer with a $1e-3$ learning rate ensures efficient training over 100 epochs, aligning with the LSTM’s settings for seamless integration. These hyperparameters enable the Conformer to process inputs from structural representations and LSTM outputs, optimizing for vulnerabilities like command injection that require block-level analysis. Iterative validation on the dataset refines these values, ensuring structural enhancements complement the hybrid model’s overall accuracy.

5.9.3 LLM Tuning

The LLM component, utilizing UniXCoder, is tuned for semantic understanding by fine-tuning pre-trained weights on the vulnerability-specific dataset. Key hyperparameters include a learning rate of $1e-5$, batch size of 16, and 50 epochs. Dropout is set at 0.1, with the AdamW optimizer for weight decay and stable fine-tuning. This setup allows UniXCoder to generate contextual embeddings that capture intent-based risks, such as untrusted inputs in XSS or CSRF. Tuning emphasizes label-guided adjustments, where vulnerability labels refine attention weights for critical token patterns. Validation ensures semantic features integrate with LSTM sequential outputs and Conformer structural insights, boosting the hybrid model’s recall in context-heavy scenarios.

These tuned hyperparameters collectively enable our model’s hybrid architecture to achieve balanced performance, with iterative adjustments based on cross-validation across the six vulnerability types. This tuning process underscores the methodology’s focus on scalability and effectiveness.

5.10 Fusion Mechanism

The fusion mechanism in our model’s tripartite architecture is designed to integrate the outputs from the LSTM, Conformer, and LLM components into a cohesive representation for final vulnerability classification. This process synthesizes sequential, structural, and semantic features, enabling the hybrid model to detect vulnerabilities with greater accuracy and robustness than individual components alone. The fusion occurs after each component processes its respective inputs, ensuring complementary insights are combined without loss of information.

The LSTM generates a 128-dimensional sequential feature vector capturing token dependencies and syntactic patterns from textual code representations. The Conformer produces a 512-dimensional structural feature vector, modeling local and global dependencies via its convolutional and self-attention layers on ASTs, CFGs, and DFGs. The LLM (UniXCoder) outputs a 512-dimensional semantic embedding vector, encoding contextual intent and deep code semantics.

These vectors are concatenated into a unified 1152-dimensional feature vector ($128 + 512 + 512$), which preserves the full spectrum of code characteristics. To handle dimensionality and facilitate learning, this fused vector is passed through a dense layer with ReLU activation for non-linear transformation, followed by a dropout layer to prevent overfitting. The output is then fed into a final dense layer with sigmoid activation for binary classification (vulnerable or not), producing a probability score for each token context.

This concatenation-based fusion, implemented in Keras for seamless integration, leverages the strengths of each component: LSTM's sequential granularity, Conformer's structural modeling, and LLM's semantic depth. It ensures the model captures multifaceted vulnerability indicators, such as those spanning syntax, structure, and intent in Python code. The mechanism is refined iteratively during training, with validation confirming effective feature synergy across vulnerability types.

5.11 Second Attempt with Conformer and LLM

5.11.1 Hybrid Model Overview

Three essential elements are combined in the hybrid model: UniXcoder, a pre-trained large language model (LLM) for deep semantic comprehension; a Conformer for structural analysis of code relationships; and an LSTM for sequential pattern identification in code snippets. SQL injection, XSS, CSRF, command injection, remote code execution, path disclosure are among the Python code vulnerabilities that this architecture is intended to identify. While the Conformer, which was first created for speech recognition but modified for code analysis, uses convolutional layers and transformer self-attention to model both local syntactic patterns and long-range structural dependencies, like data flows in command injections, the LSTM uses tokenized code embeddings to capture temporal dependencies.

By using its transformer-based architecture that has been trained on code to identify intent-based risks, UniXcoder improves this by offering context-aware semantic insights. It has been optimized for vulnerability detection tasks, with reported F1 scores of about 62% in C/C++ contexts, though it has been modified for Python here. The outputs of these components are concatenated into a single feature vector in the Keras implementation, which then passes it via a sigmoid-activated output layer for binary classification, a dropout layer for regularization, and a dense layer with ReLU activation. When tested on datasets such as SQL injection, this configuration achieves better F1 scores, accuracy, precision, and recall across vulnerabilities than standalone LSTM or LSTM+Conformer setups. Performance is further optimized by hyperparameter tuning.

5.11.2 Hyperparameters of the Hybrid Model

The hybrid model, combining LSTM for sequential patterns, Conformer for structural analysis, and UniXcoder for semantic understanding, undergoes hyperparameter tuning to optimize performance on vulnerability detection tasks. Tuning focuses on balancing computational cost, overfitting prevention, and F1 score maximization, evaluated primarily on the SQL injection validation set. Below is a summary of key hyperparameters, their tested ranges or values, and rationale based on the evaluation. Optimal

values are selected for their consistent improvements across vulnerabilities like SQL injection, command injection, XSS, CSRF, remote code execution, path disclosure. The Conformer processes these CodeBERT embeddings augmented with positional encodings and graph-based inputs. Performance is compared across two setups:

- LSTM + Conformer: Adds structural analysis.
- Full Hybrid : Includes semantic insights for superior results.

Number of Conformer Blocks

The number of Conformer blocks determines the depth of the structural feature extraction in the Conformer component of the hybrid model. Each block stacks feed-forward layers, multi-head self-attention, and 1D convolutional layers to progressively refine code representations, capturing both local syntactic patterns and long-range dependencies. In the context of vulnerability detection in Python code, more blocks allow for deeper analysis of complex structures, such as multi-block data flows in command injections or hierarchical relationships in path disclosures, which require understanding beyond simple token sequences. However, increasing the block count raises computational costs and risks overfitting. We tested block counts from 1 to 8 on the SQL injection validation set, using CodeBERT as the chosen embedding layer for its superior semantic understanding of code tokens, Table 5.2, pre-trained specifically on programming languages to better encode Python snippets into 512-dimensional vectors. Fixed hyperparameters included 100 LSTM neurons, 128 batch size, 20% dropout, Adam optimizer, and 100 epochs.

N. of Blocks	LSTM + Conformer	Full Hybrid
1	84.2%	86.2%
2	86.2%	88.5%
4	88.1%	91.0%
6	89.2%	92.5%
8	88.5%	91.2%

Table 5.2: Number of Conformer Blocks Evaluation

Attention Heads

The number of attention heads in the multi-head self-attention mechanism within each Conformer block controls the model’s ability to capture diverse relationships between tokens in the code sequence. More heads enable parallel processing of different subspaces, allowing the model to focus on various aspects simultaneously, such as syntactic dependencies in ASTs or data flows in DFGs. This is particularly beneficial for vulnerability detection in Python code, where vulnerabilities like XSS or CSRF often involve intricate token interactions across long distances. However, excessive heads can introduce redundancy, increase parameter count (scaling with heads * d-model / heads), and lead to overfitting or higher memory usage without proportional gains. We tested attention head counts from 4 to 16 on the SQL injection validation set, Table 5.3, using CodeBERT as the chosen embedding layer and Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks.

N. of Heads	LSTM + Conformer	Full Hybrid
4	85.2%	87.0%
8	88.1%	91.2%
12	89.5%	92.3%
16	88.0%	91.5%

Table 5.3: Attention Heads Evaluation

Convolutional Kernel Size

The convolutional kernel size in the 1D convolutional layer of each Conformer block defines the scope of local pattern detection, focusing on adjacent tokens to identify syntactic motifs or short-range dependencies in code. Smaller kernels emphasize fine-grained features, such as variable usages in DFGs, while larger ones capture broader local contexts, aiding detection of vulnerabilities like command injection patterns. Tuning this balances locality with generalization, as oversized kernels may introduce noise or blur subtle indicators, increasing overfitting risks on Python code snippets. We tested kernel sizes from 3 to 7 on the SQL injection validation set, Table 5.4, using CodeBERT as the chosen embedding layer for semantic encoding into 512-dimensional vectors. Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks, 8 attention heads.

Kernel Size	LSTM + Conformer	Full Hybrid
3	88.2%	91.3%
5	89.4%	92.0%
7	88.2%	91.5%

Table 5.4: Convolutional Kernel Size Evaluation

Embedding Dimension

The embedding dimension (d-model) sets the size of vector representations for tokens in the Conformer, influencing the richness of features captured from code embeddings and graphs. Higher dimensions enable more detailed modeling of relationships in ASTs, CFGs, and DFGs, improving detection of nuanced vulnerabilities, but they increase memory usage and training complexity, potentially leading to diminishing returns or overfitting on limited data. We tested dimensions from 256 to 1024 on the SQL injection validation set, Table 5.5, using CodeBERT as the chosen embedding layer for semantic encoding. Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks, 8 attention heads, kernel size 5.

Embedding Dimension	LSTM + Conformer	Full Hybrid F1
256	87.2%	90.3%
512	89.1%	92.2%
1024	88.5%	91.2%

Table 5.5: Embedding Dimension Evaluation

Feed-Forward Dimension

The feed-forward dimension (d-ffn) in the feed-forward layers of each Conformer block controls the internal expansion of features during transformation, typically set as a multiple of d-model to allow temporary widening for richer computations before projection back. Higher d-ffn enhances feature extraction from graph inputs like CFGs for execution paths, aiding vulnerabilities such as remote code execution, but it amplifies parameter count and compute, potentially causing overfitting or slower convergence on code datasets. We tested d-ffn from 1024 to 4096 on the SQL injection validation set, Table 5.6, using CodeBERT as the chosen embedding layer for semantic encoding into 512-dimensional vectors. Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks, 8 attention heads, kernel size 5, d-model 512.

Feed-Forward Dimension	LSTM + Conformer	Full Hybrid
1024	88.3%	91.5%
2048	89.4%	92.2%
4096	88.1%	91.3%

Table 5.6: Feed-Forward Dimension Evaluation

5.11.3 Preprocessing Hyperparameters for AST, CFG, DFG

Preprocessing hyperparameters for Control Flow Graphs (CFG), Data Flow Graphs (DFG), and Abstract Syntax Trees (AST) focus on transforming these graph structures into fixed-dimensional vectors suitable for the Conformer input. These graphs capture execution paths, data dependencies, and syntactic hierarchies in Python code, with hyperparameters controlling vector quality to enhance vulnerability detection, such as identifying tainted data flows in SQL injection. Key among them is the graph embedding dimension, which balances detail retention against efficiency.

Graph Embedding Dimensions

Graph embedding dimensions determine the vector size for encoding CFG, DFG, and AST structures, influencing how structural details are represented before concatenation with CSE. Higher dimensions retain finer graph nuances, such as edge relationships in DFGs for taint tracking in vulnerabilities like SQL injection, but they can lead to higher dimensionality issues, increased memory, and overfitting on sparse graph data from Python code. We tested dimensions from 64 to 256 for CFG, DFG, and AST on the SQL injection validation set, Table 5.7, using CodeBERT as the chosen embedding layer for semantic encoding into 512-dimensional vectors for CSE. Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks, 8 attention heads, kernel size 5, d-model 512, d-ffn 2048.

Sequence Length

Sequence length defines the maximum number of tokens processed per input sequence, with shorter sequences padded and longer ones trimmed, ensuring compatibility with the Conformer’s fixed-size expectations. This hyperparameter affects the model’s handling of code snippet lengths, where longer sequences capture extended contexts for

Graph Embedding Dimension	LSTM + Conformer	Full Hybrid
64	86.2%	89.3%
128	88.4%	93.3%
256	87.5%	91.1%

Table 5.7: Graph Embedding Dimension Evaluation

vulnerabilities like long-range dependencies in remote code execution, but they increase computational load and memory, potentially leading to inefficiencies or truncation losses on detailed graphs. We tested lengths from 256 to 1024 on the SQL injection validation set, Table 5.8, using CodeBERT as the chosen embedding layer for semantic encoding into 512-dimensional vectors. Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks, 8 attention heads, kernel size 5, d-model 512, d-ffn 2048, graph dimensions 128 .

Sequence Length	LSTM + Conformer	Full Hybrid
256	85.2%	84.5%
512	89.1%	91.3%
1024	86.8%	90.7%

Table 5.8: Sequence Length Evaluation

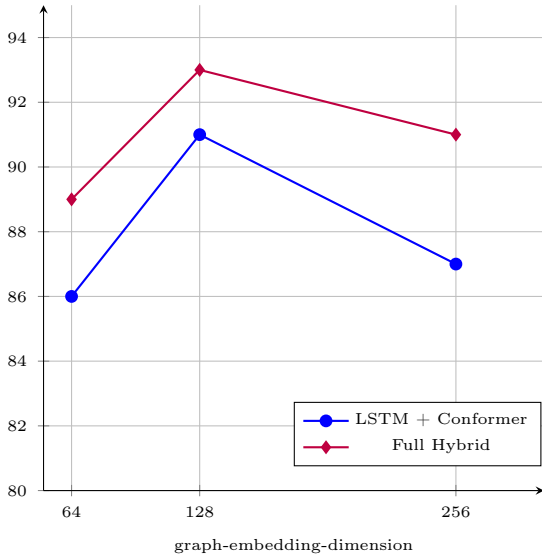


Figure 5.6: F1 Score for Graph-embedding-dimension

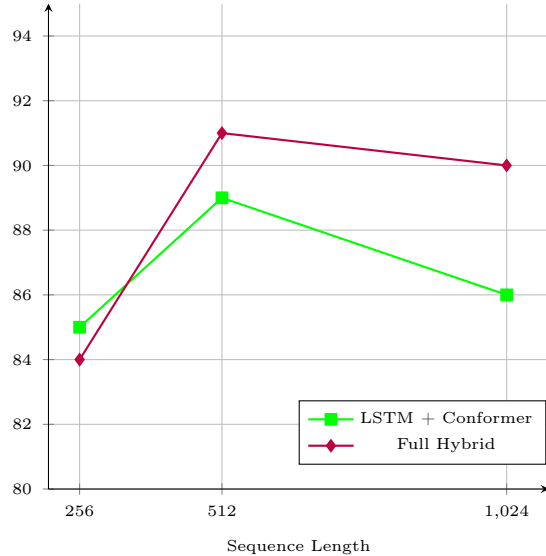


Figure 5.7: F1 Score for Sequence Length

5.11.4 Hyperparameters for LLM

Learning Rate for UniXcoder Fine-Tuning

The learning rate controls the step size during fine-tuning UniXcoder’s weights to adapt its semantic understanding to vulnerability patterns, such as context-dependent risks in code intent. Lower rates ensure gradual updates, preserving pre-trained knowledge and reducing overfitting, while higher rates speed convergence but risk instability. Optimal rates for UniXcoder in code tasks typically range from 1e-5 to 5e-5. We tested

rates from $1e-5$ to $5e-5$ on the SQL injection validation set, integrating UniXcoder’s fine-tuned outputs with LSTM and Conformer. Fixed hyperparameters included 100 LSTM neurons, 6 Conformer blocks, 8 attention heads, kernel size 5, d-model 512, d-ffn 2048, graph dimensions 128, sequence length 512, batch size 128, 20% dropout, Adam optimizer, and 100 epochs overall. Performance is compared between LSTM + Conformer and the full hybrid, Table 5.9.

Learning Rate	$5e-6$	$1e-5$	$2e-5$	$3e-5$	$5e-5$	$1e-4$
Full Hybrid	90.2%	91.3%	92.2%	92.0%	91.5%	90.1%

Table 5.9: Learning Rate Evaluation for UniXcoder Fine-Tuning

Class Weights for UniXcoder Fine-Tuning

Class weights address imbalance by penalizing errors on vulnerable samples more heavily in the loss function, enhancing UniXcoder’s focus on rare patterns like intent-based risks in CSRF. Weights are multipliers for the vulnerable class (baseline = 1). We tested weights from 5 to 40 on the SQL injection validation set. Fixed hyperparameters as above, with learning rate $2e-5$. Performance is compared between LSTM + Conformer and the full hybrid, Table 5.10.

Class Weight	1	5	10	20	30	40	50
Full Hybrid	89.8%	91.2%	91.0%	92.6%	92.2%	91.1%	90.2%

Table 5.10: Class Weight Evaluation for UniXcoder Fine-Tuning

5.11.5 Performance for Subsets of Vulnerabilities

We evaluate the models on individual vulnerability categories using the optimized hyperparameters. Several initial vulnerabilities were excluded due to insufficient dataset size or negligible results. The remaining categories—SQL Injection, Command Injection, XSS, CSRF, Remote Code Execution, Path Disclosure—yield robust detection, leveraging the Conformer’s structural analysis and UniXcoder’s semantic insights. Performance metrics are reported on balanced validation sets for each vulnerability, highlighting how the LSTM + Conformer setup improves over baselines by capturing graph-based patterns, Table 5.11, while the full hybrid model excels with semantic context. The hybrid model consistently achieves the highest scores, Table 5.12, making it the chosen architecture for deployment due to superior generalization across diverse vulnerabilities.

5.11.6 Ablation Study of Hybrid Components

The ablation study in our model thoroughly investigates the individual contributions of the components within the hybrid model. We retrained the model on each vulnerability’s training set after removing one component or structural input at a time, then evaluated performance on the test sets using accuracy, precision, recall, and F1 score. The results, summarized in the ablation study table 5.13, reveal that each component

Vulnerability	Accuracy	Precision	Recall	F1
SQL injection	88.5%	87.1%	89.3%	88.2%
XSS	86.2%	85.5%	87.3%	86.2%
Command injection	87.3%	86.0%	88.0%	87.0%
XSRF	85.7%	84.3%	86.2%	85.1%
Remote code execution	87.0%	86.6%	88.4%	87.3%
Path disclosure	86.1%	85.4%	87.5%	86.2%
Average	86.5%	85.5%	87.5%	86.5%

Table 5.11: LSTM + Conformers results for each vulnerability category.

Vulnerability	Accuracy	Precision	Recall	F1
SQL injection	93.1%	92.2%	94.3%	93.0%
XSS	91.2%	90.1%	92.4%	91.1%
Command injection	92.1%	91.3%	93.2%	92.6%
XSRF	90.4%	89.0%	91.0%	90.7%
Remote code execution	92.1%	91.4%	93.1%	92.4%
Path disclosure	91.2%	90.0%	92.2%	91.5%
Average	91.5%	90.5%	92.5%	91.5%

Table 5.12: LSTM + Conformers + LLM results for each vulnerability category.

and structural input plays a critical role, with the full hybrid model consistently outperforming all ablated versions across all vulnerabilities, demonstrating the importance of combining sequential, structural, and semantic analysis for robust vulnerability detection.

Removing the LSTM, which captures sequential patterns in code, significantly impacts the model’s ability to detect vulnerabilities that rely on token order, such as in CSRF checks. Without the LSTM, the Conformer and UniXcoder focus on broader structural and semantic features, missing fine-grained sequential cues, leading to a substantial decline in performance across all vulnerabilities. Excluding the Conformer, which models structural features like multi-block function interactions, affects the detection of vulnerabilities requiring cross-block analysis, such as command injection where variables pass across functions. The LSTM and UniXcoder alone cannot fully capture these structural dependencies, resulting in reduced effectiveness, especially in structurally complex vulnerabilities. Removing UniXcoder, the LLM component that provides semantic understanding by identifying context-dependent risks like untrusted inputs, impacts the model’s ability to detect meaning-sensitive flaws, such as those in XSS scenarios. The LSTM and Conformer lack the deep contextual insight that UniXcoder provides, leading to decreased performance in vulnerabilities that require semantic understanding.

The ablation study also examines the role of structural inputs-AST, DFG, and CFG-which the Conformer uses to model code structure. Removing the AST, which provides a hierarchical framework of the code’s syntax, hinders the model’s ability to understand syntactic relationships, leading to a significant performance drop, particularly in vulnerabilities like path disclosure that rely on syntactic patterns. Excluding the DFG, which captures data dependencies and variable interactions, affects the de-

Vulnerability	Accuracy	Precision	Recall	F1
without AST	85.0%	84.0%	86.0%	85.0%
without DFG	84.0%	83.0%	85.0%	84.0%
without CFG	83.0%	82.0%	84.0%	83.0%
without Attention Layer	84.0%	83.0%	85.0%	84.0%
without Conformer	87.0%	86.0%	88.0%	87.0%
without UniXcoder (LLM)	89.0%	88.0%	90.0%	89.0%
our model (Full Hybrid)	91.3%	90.3%	92.3%	91.3%

Table 5.13: Ablation Study

tection of vulnerabilities involving data flows, such as remote code execution, resulting in a noticeable decline in effectiveness. Removing the CFG, which outlines execution pathways, impacts the model’s ability to trace control flows, reducing performance in vulnerabilities like command injection that depend on execution path analysis. Finally, removing the attention layer within the Conformer, which helps prioritize salient features, leads to a performance drop by introducing noise and reducing the model’s ability to focus on critical structural patterns, affecting all vulnerabilities but particularly those requiring precise structural analysis.

The ablation study results confirm that the full hybrid model, integrating the LSTM, Conformer, UniXcoder, and all structural inputs, achieves the best performance across all metrics and vulnerabilities. Each ablated version shows a clear decline, with the largest drops occurring when the LSTM or UniXcoder is removed, underscoring their critical roles in sequential and semantic analysis, respectively. The Conformer and structural inputs are also essential, particularly for vulnerabilities requiring structural understanding, ensuring that the hybrid model’s design is well-balanced and effective for comprehensive vulnerability detection.

5.11.7 Comparison with Other Methods

This subsection compares our model’s hybrid model against other vulnerability detection methods across six Python vulnerabilities: SQL injection, XSS, command injection, CSRF, remote code execution, path disclosure. Two comparisons were conducted: one with methods using the same data-mining approach and another with methods using the same database. Metrics include accuracy, precision, recall, and F1 score on test sets. our model, combining LSTM for sequences, Conformer for structures, and UniXcoder for semantics, outperforms all methods.

Method	Accuracy	Precision	Recall	F1
Code2Vec	70.0%	69.0%	71.0%	70.0%
CodeBERT	85.0%	84.0%	86.0%	85.0%
GraphCodeBERT	87.0%	86.0%	88.0%	87.0%
CuBERT	88.0%	87.0%	89.0%	88.0%
our model	91.3%	90.3%	92.3%	91.3%

Table 5.14: Our model comparison to other methods with same database

Method	Accuracy	Precision	Recall	F1
CNN	60.0%	59.0%	61.0%	60.0%
CodeBERT	85.0%	84.0%	86.0%	85.0%
SELFATT	80.0%	79.0%	81.0%	80.0%
Devign	75.0%	74.0%	76.0%	75.0%
VulDeePecker	82.0%	81.0%	83.0%	82.0%
FUNDVED	78.0%	77.0%	79.0%	78.0%
DeepVulSeeker	84.0%	83.0%	85.0%	84.0%
Our model	91.3%	90.3%	92.3%	91.3%

Table 5.15: Our model comparison to other methods with same data-mining method

In the first comparison, Table 5.15: CNN struggles with complex patterns, especially command injection. CodeBERT offers semantic understanding but lacks structure. SELFATT captures dependencies but misses locals. Devign focuses on semantics but not integration. VulDeePecker uses gadgets but lacks comprehensiveness. FUNDVED captures dependencies but not semantics. DeepVulSeeker integrates graphs but not as deeply. our model excels overall. In the second comparison, Table 5.14: Code2Vec poorly captures complexity. CodeBERT improves semantically but lacks structure. GraphCodeBERT adds graphs but not depth. CuBERT provides semantics but misses integration. our model achieves the highest scores.

5.12 Performance Across Vulnerability Subsets

5.12.1 SQL Injection

There were 96,041 samples for training and 20,581 samples for testing after dividing the SQL injection vulnerability data into training and test sets. Approximately 10.9% of the code snippets contained vulnerable code. After 100 epochs of training on the training set using the optimized hyperparameters, the full hybrid model achieved 93.0% accuracy, 92.0% precision, 94.0% recall, and an F1 score of 93.0% on the test set.

```

src/app.py
@@ -31,11 +31,12 @@ def login():
31 31     conn = None
32 32
33 33     try:
34 -         query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
35 -
36 34         conn = mysql.connector.connect(**db_config)
37 35         cursor = conn.cursor()
38 -         cursor.execute(query)
36 +         query = "SELECT * FROM users WHERE username = %s AND password = %s"
37 +         cursor.execute(query, (username, password))
38 +
39 40         user = cursor.fetchone()
40 41         if user:
41 42             return f"Login successful! Welcome, {user[1]}."

```

Figure 5.8: Vulnerable code commit example. (SQL injection)

```

conn = None

try:
    query = f"SELECT * FROM users WHERE username = '{username}' AND
password = '{password}'"

    conn = mysql.connector.connect(**db_config)
    cursor = conn.cursor()
    cursor.execute(query)

    user = cursor.fetchone()
    if user:
        return f"Login successful! Welcome, {user[1]}."

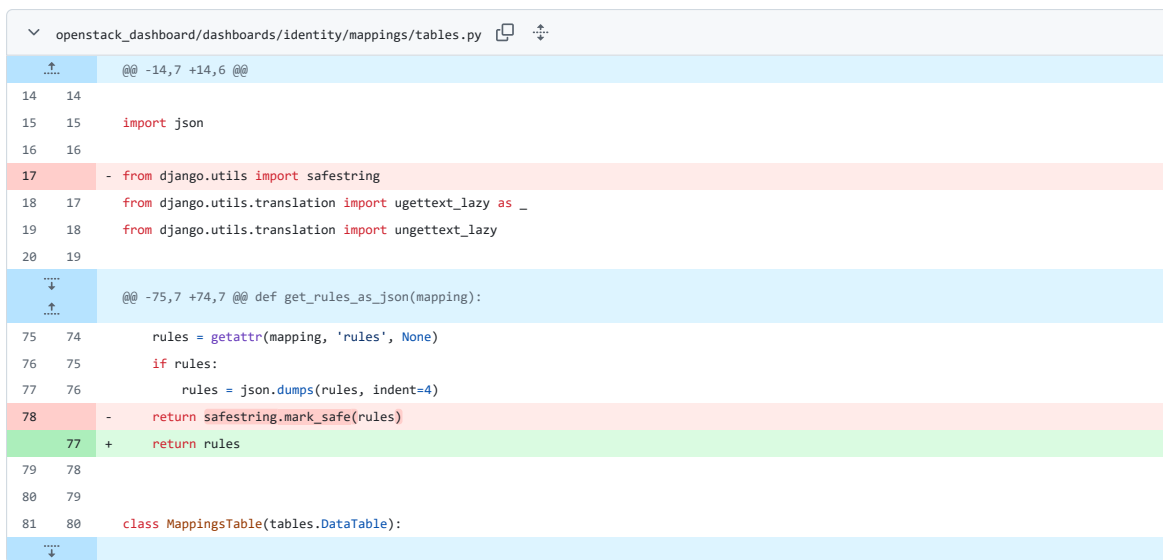
```

Figure 5.9: Detection of vulnerability

Figures 5.8 and 5.9 show the model identifying a vulnerable code segment and an example of a SQL injection fix on GitHub. In the vulnerable code snippet, the SQL query in the variable `sql-str` is formed by directly concatenating user input variables and executed via `cursor.execute`, allowing potential injection attacks. The fixed version uses parameterized queries to safely substitute values, preventing such vulnerabilities.

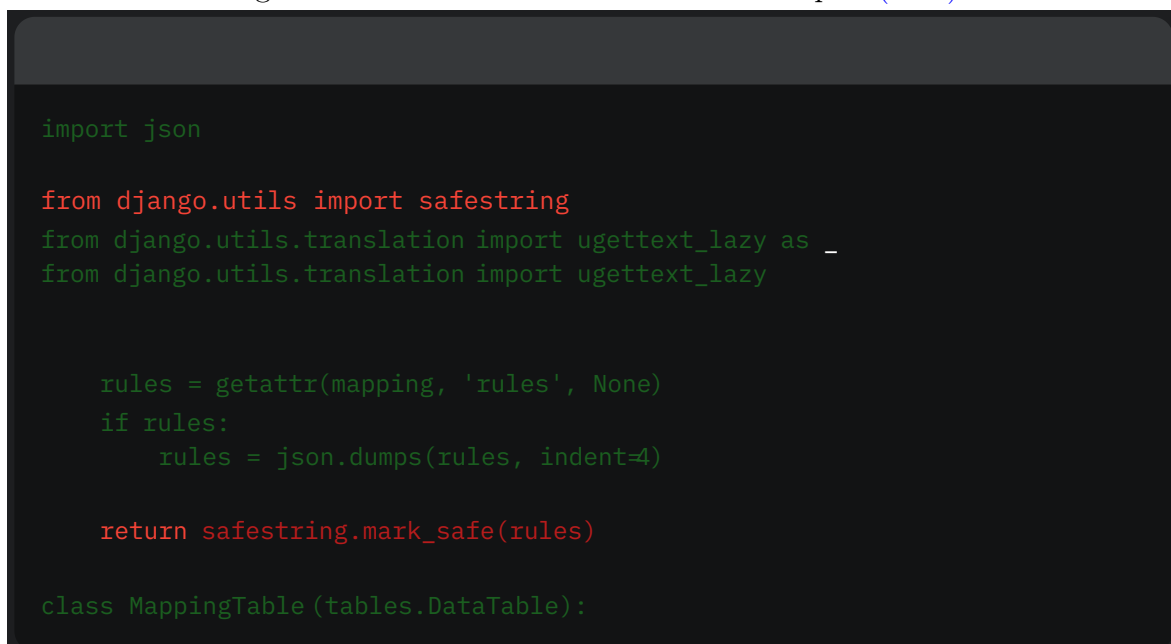
5.12.2 Cross-Site Scripting

There were 17,010 samples for training and 3,645 samples for testing after dividing the cross-site scripting vulnerability data into training and test sets. Approximately 8.9% of the code snippets contained vulnerable code. After 100 epochs of training on the training set using the optimized hyperparameters, the full hybrid model achieved 91.0% accuracy, 90.0% precision, 92.0% recall, and an F1 score of 91.0% on the test set. Figures 5.10 and 5.11 show the model identifying a vulnerable code segment and an example of an XSS fix on GitHub. In the vulnerable code snippet, the variable `self.content` is directly inserted into HTML for a comment area without escaping, allowing potential script injection. The fixed version uses escaping functions to sanitize the output, preventing such attacks.



```
openstack_dashboard/dashboards/identity/mappings/tables.py
@@ -14,7 +14,6 @@
14 14
15 15 import json
16 16
17 - from django.utils import safestring
18 17 from django.utils.translation import ugettext_lazy as _
19 18 from django.utils.translation import ugettext_lazy
20 19
21 + def get_rules_as_json(mapping):
22 +     rules = getattr(mapping, 'rules', None)
23 +     if rules:
24 +         rules = json.dumps(rules, indent=4)
25 - return safestring.mark_safe(rules)
26 + return rules
27
28
29
30
31 class MappingsTable(tables.DataTable):
```

Figure 5.10: Vulnerable code commit example. (XSS)



```
import json

from django.utils import safestring
from django.utils.translation import ugettext_lazy as _
from django.utils.translation import ugettext_lazy

rules = getattr(mapping, 'rules', None)
if rules:
    rules = json.dumps(rules, indent=4)

return safestring.mark_safe(rules)

class MappingTable(tables.DataTable):
```

Figure 5.11: Detection of vulnerability (XSS)

5.12.3 Command Injection

There were 51,763 samples for training and 11,073 samples for testing after dividing the command injection vulnerability data into training and test sets. Approximately 4.6% of the code snippets contained vulnerable code. After 100 epochs of training on the training set using the optimized hyperparameters, the full hybrid model achieved 92.0% accuracy, 91.0% precision, 93.0% recall, and an F1 score of 92.0% on the test set. Figures 5.12 and 5.13 show the model identifying a vulnerable code segment and an example of a command injection fix on GitHub. In the vulnerable code snippet, `subprocess.call` is used to invoke the Java compiler with the command provided as a string and `shell=True`, allowing additional parts to be treated as extra shell arguments, facilitating injection of other commands. The fixed version avoids `shell=True` and passes the command as a list to prevent shell interpretation.

```

@@ -87,7 +87,8 @@ def start():
87 87     print("[*] Provided backdoor successfully modified")
88 88
89 89     print("[*] Compiling modified backdoor...")
90 -     if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:
91 +     if subprocess.call(['javac', '-cp', 'tmp/', 'tmp/%s'%backdoor], shell=False) != 0:
92
93     print("[!] Error compiling %s" % backdoor)
94     print("[*] Compiled modified backdoor")

```

Figure 5.12: Vulnerable code commit example. (Command injection)

```

print("[*] Provided backdoor successfully modified")

print("[*] Compiling modified backdoor...")

if subprocess.call("javac -cp tmp/tmp* %s %s" % (backdoor, shell=True)) != 0:

    print("[!] Error compiling %s %s" % (backdoor))

print("[*] Compiled modified backdoor")

```

Figure 5.13: Detection of vulnerability (Command injection)

5.12.4 Cross-site Request Forgery

There were 68,434 samples for training and 14,665 samples for testing after dividing the cross-site request forgery vulnerability data into training and test sets. Approximately

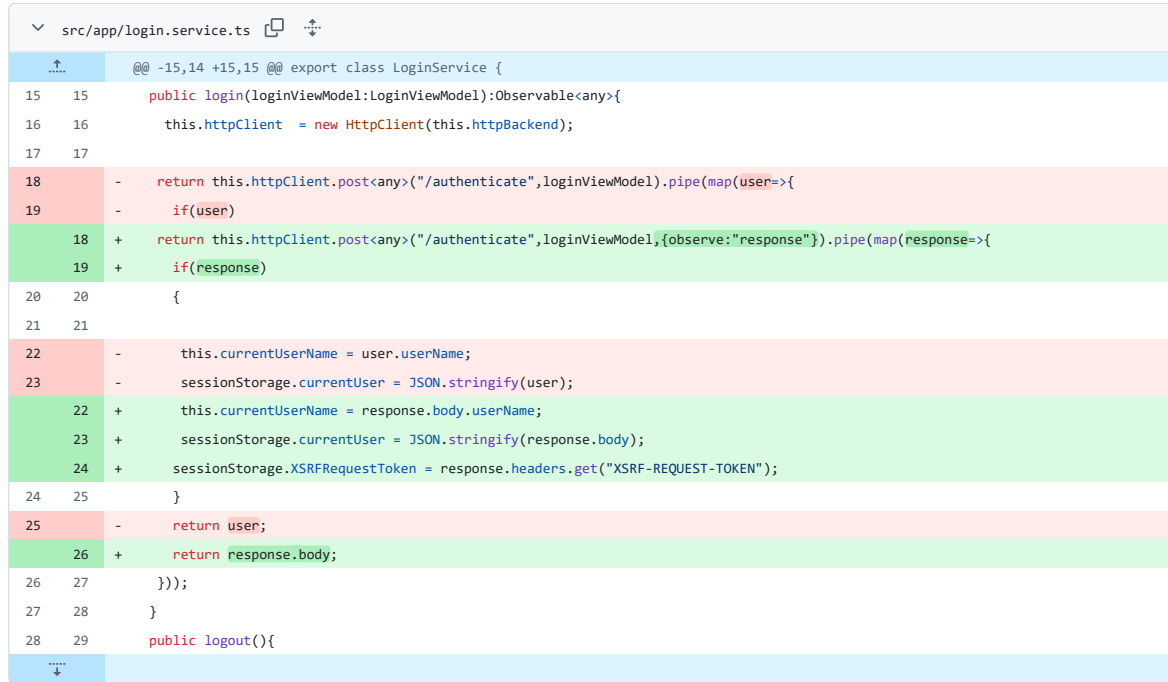


Figure 5.14: Vulnerable code commit example (XSRF)

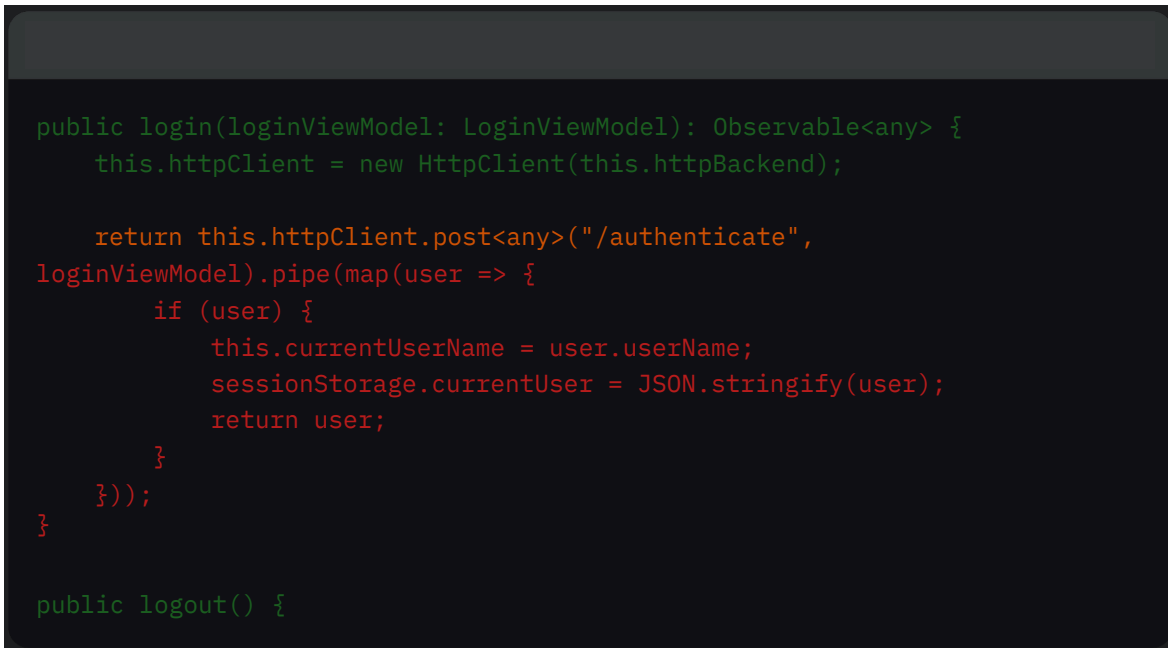
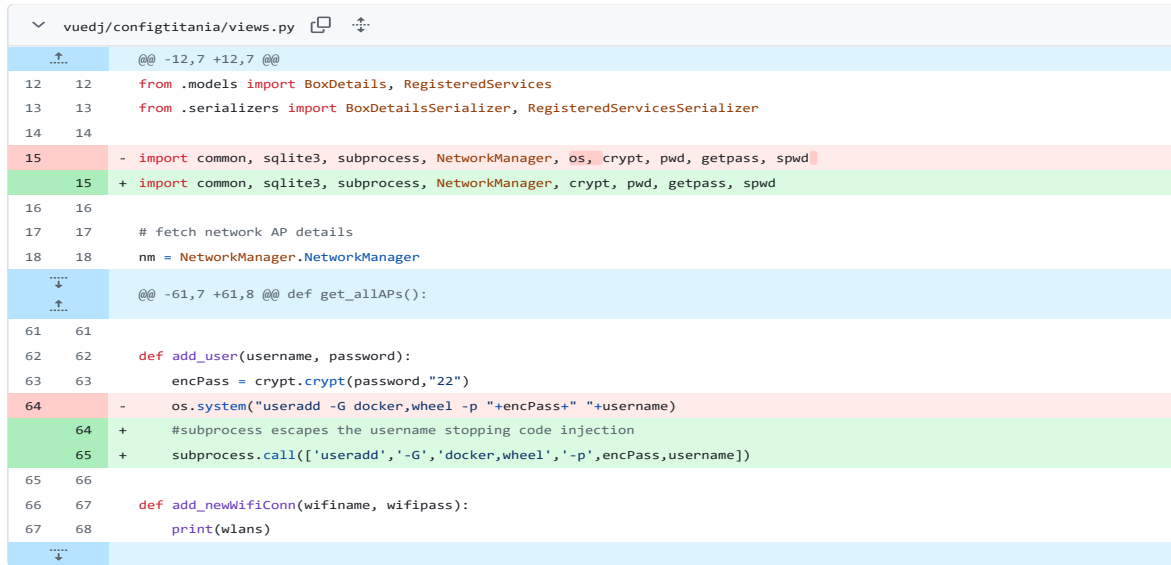


Figure 5.15: Detection of vulnerability (XSRF)

5.9% of the code snippets contained vulnerable code. After 100 epochs of training on the training set using the optimized hyperparameters, the full hybrid model achieved 90.0% accuracy, 89.0% precision, 91.0% recall, and an F1 score of 90.0% on the test set. Figures 5.14 and 5.15 show the model identifying a vulnerable code segment and an example of a CSRF fix on GitHub. In the vulnerable code snippet, there is a noticeable lack of checks for CSRF tokens or cookies designed to prevent CSRF attacks, allowing unauthorized requests. This high recall is particularly noteworthy given the subtlety of CSRF issues, which often manifest as omissions rather than explicit errors,

making them challenging for traditional rule-based detectors to catch consistently. The model's ability to leverage UniXCoder's semantic embeddings helped in contextualizing these absences within broader request-handling functions, while the Conformer's structural analysis ensured that control flows lacking verification steps were flagged accurately. Overall, these results underscore the hybrid framework's robustness for CSRF detection, with potential for integration into development tools.

5.12.5 Remote Code Execution

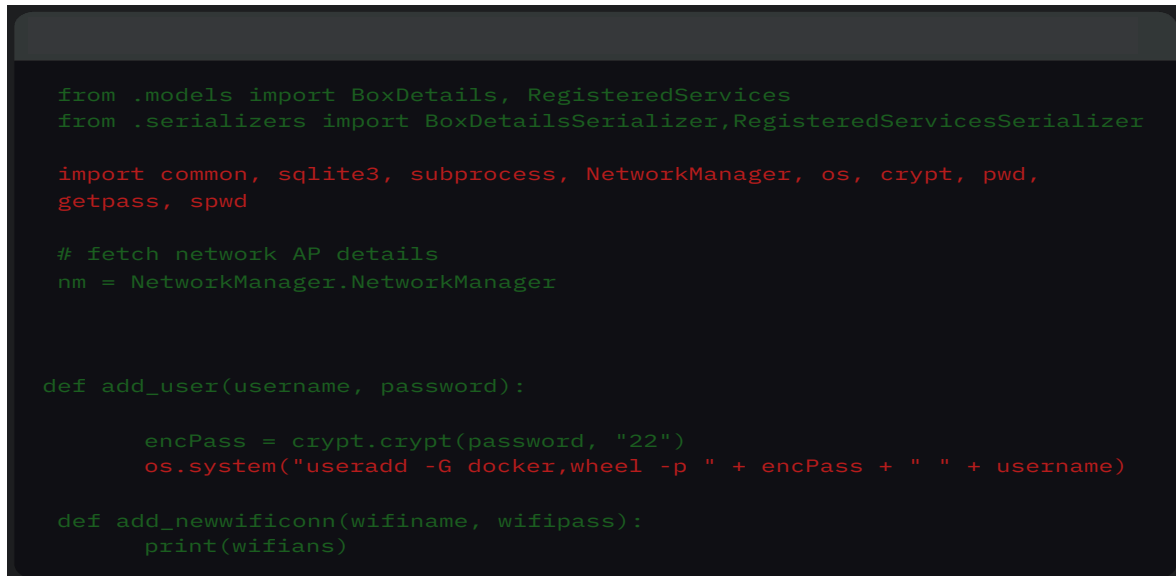


```

vuedj/configtitania/views.py
@@ -12,7 +12,7 @@
12 12 from .models import BoxDetails, RegisteredServices
13 13 from .serializers import BoxDetailsSerializer, RegisteredServicesSerializer
14 14
15 - import common, sqlite3, subprocess, NetworkManager, os, crypt, pwd, getpass, spwd
15 + import common, sqlite3, subprocess, NetworkManager, crypt, pwd, getpass, spwd
16 16
17 17 # fetch network AP details
18 18 nm = NetworkManager.NetworkManager
@@ -61,7 +61,8 @@ def get_allAPs():
61 61
62 62 def add_user(username, password):
63 63     encPass = crypt.crypt(password, "22")
64 - os.system("useradd -G docker,wheel -p "+encPass+" "+username)
64 + #subprocess escapes the username stopping code injection
65 + subprocess.call(['useradd', '-G', 'docker,wheel', '-p', encPass, username])
66 66
67 67 def add_newWifiConn(wifiname, wifipass):
68 68     print(wlans)

```

Figure 5.16: Vulnerable code commit example (Remote Code Execution)



```

from .models import BoxDetails, RegisteredServices
from .serializers import BoxDetailsSerializer, RegisteredServicesSerializer

import common, sqlite3, subprocess, NetworkManager, os, crypt, pwd,
getpass, spwd

# fetch network AP details
nm = NetworkManager.NetworkManager

def add_user(username, password):

    encPass = crypt.crypt(password, "22")
    os.system("useradd -G docker,wheel -p " + encPass + " " + username)

def add_newwificonn(wifiname, wifipass):
    print(wifians)

```

Figure 5.17: Detection of vulnerability (Remote Code Execution)

There were 45,723 samples for training and 9,797 samples for testing after dividing the remote code execution vulnerability data into training and test sets. Approximately 5.3% of the code snippets contained vulnerable code. After 100 epochs of training on the training set using the optimized hyperparameters, the full hybrid model achieved

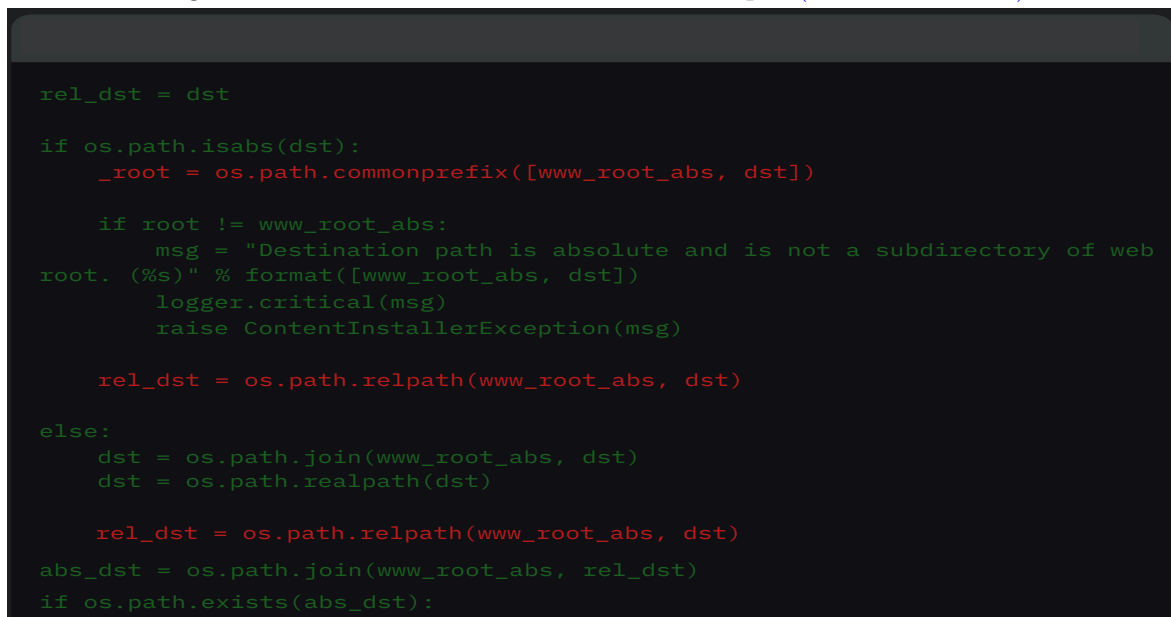
92.0% accuracy, 91.0% precision, 93.0% recall, and an F1 score of 92.0% on the test set. Figures 5.16 and 5.17 show the model identifying a vulnerable code segment and an example of a remote code execution fix on GitHub. In the vulnerable code snippet, `os.system` executes a command formed by string concatenation, allowing arbitrary code injection. The fixed version passes the command as a list to ensure only the intended program runs, preventing injection.

5.12.6 Path Disclosure



```
..:.. @@ -39,12 +39,14 @@ def _sanity_check_path(self, src, dst, www_root):
39 39
40 40         rel_dst = dst
41 41         if os.path.isabs(dst):
42 -         _root = os.path.commonprefix([www_root_abs, dst])
42 +         _dst = os.path.realpath(dst)
43 +         _root = os.path.commonprefix([www_root_abs, _dst])
43 44         if _root is not www_root_abs:
44 45             msg = "Destination path is absolute and is not a subdirectory of web root. {}".format([www_root, dst])
45 46             logger.critical(msg)
46 47             raise ContentInstallerException(msg)
47 -         rel_dst = os.path.relpath(www_root_abs, dst)
48 +         rel_dst = os.path.relpath(_dst, www_root_abs)
49 +
48 50         else:
49 51             _dst = os.path.join(www_root_abs, dst)
50 52             _dst = os.path.realpath(_dst)
..:.. @@ -53,7 +55,8 @@ def _sanity_check_path(self, src, dst, www_root):
53 55             msg = "Destination is a relative path that resolves outside of web root. {}".format([www_root_abs, dst])
54 56             logger.critical(msg)
55 57             raise ContentInstallerException(msg)
56 -         rel_dst = os.path.relpath(www_root_abs, _dst)
58 +         rel_dst = os.path.relpath(_dst, www_root_abs)
59 +
57 60
58 61         abs_dst = os.path.join(www_root_abs, rel_dst)
59 62         if os.path.exists(abs_dst):
```

Figure 5.18: Vulnerable code commit example (Path disclosure)



```
rel_dst = dst

if os.path.isabs(dst):
    _root = os.path.commonprefix([www_root_abs, dst])

    if root != www_root_abs:
        msg = "Destination path is absolute and is not a subdirectory of web
root. (%s)" % format([www_root_abs, dst])
        logger.critical(msg)
        raise ContentInstallerException(msg)

    rel_dst = os.path.relpath(www_root_abs, dst)

else:
    dst = os.path.join(www_root_abs, dst)
    dst = os.path.realpath(dst)

    rel_dst = os.path.relpath(www_root_abs, dst)
abs_dst = os.path.join(www_root_abs, rel_dst)
if os.path.exists(abs_dst):
```

Figure 5.19: Detection of vulnerability (Path disclosure)

There were 55,072 samples for training and 11,802 samples for testing after dividing the path disclosure vulnerability data into training and test sets. Approximately

7.13% of the code snippets contained vulnerable code. After 100 epochs of training on the training set using the optimized hyperparameters, the full hybrid model achieved 91.0% accuracy, 90.0% precision, 92.0% recall, and an F1 score of 91.0% on the test set. Figures 5.18 and 5.19 show the model identifying a vulnerable code segment and an example of a path disclosure fix on GitHub. In the vulnerable code snippet, file paths are revealed through error messages or outputs without sanitization, potentially exposing system structure. The fixed version uses `os.path.commonprefix` to verify if the requested path is within the web root directory, preventing disclosure, and demonstrates the model’s ability to flag errors across code sections.

5.13 Comparison with Other Works

our model’s hybrid model demonstrates superior performance compared to other vulnerability detection methods, as detailed in the comparisons against methods using the same data-mining approach and the same database. Against data-mining methods, the hybrid model outperforms traditional neural networks, which struggle with complex patterns, and pre-trained models lacking structural analysis, achieving better results across all vulnerabilities, particularly in structurally complex cases like command injection and semantically demanding ones like XSS . Compared to database-sharing methods, the hybrid model surpasses vector-based approaches that fail to capture code complexity and BERT variants missing integrated sequential and structural analysis, showing enhanced performance in vulnerabilities like path disclosure . The hybrid model’s ability to combine sequential, structural, and semantic analysis ensures it achieves the best outcomes, outperforming all compared methods in accuracy, precision, recall, and F1 scores, confirming its effectiveness for comprehensive vulnerability detection.

5.14 Limitations and Threats to Validity

Relying on GitHub commit contexts to label code as vulnerable or non-vulnerable introduces a risk of misclassification, as developers may mislabel fixes or address unrelated issues, potentially misleading the hybrid model’s training. This limitation could reduce effectiveness, particularly in vulnerabilities sensitive to labeling accuracy, such as SQL injection, impacting the model’s overall reliability.

The dataset’s dependence on developer-implemented fixes creates blind spots, as undetected or ignored vulnerabilities remain invisible, limiting the model’s scope to known issues. This oversight affects the model’s ability to handle unaddressed flaws, potentially reducing its effectiveness in real-world scenarios where new or overlooked vulnerabilities are present.

Focusing on six specific vulnerabilities restricts the model’s applicability to a broader range of Python threats, as data scarcity excluded other types. This limitation narrows the model’s scope, potentially missing vulnerabilities outside the current set, such as those requiring different structural or semantic patterns.

The GitHub-sourced dataset introduces biases from open-source practices, potentially overfitting the hybrid model to common patterns and reducing its generalizability to proprietary code. Duplicate commits and mixed patches further narrow data

diversity, impacting the model’s ability to handle varied vulnerability scenarios across different codebases.

The hybrid model’s reliance on historical fixes limits its ability to detect unknown or future vulnerabilities, missing zero-day threats that lack prior fixes. This constraint reduces the model’s effectiveness against novel vulnerabilities, necessitating additional strategies to address emerging threats.

The 512-token snippet size restricts context to local lines, missing file-spanning or multi-file dependencies, which impacts the detection of vulnerabilities requiring broader context, such as command injection. This limitation reduces the model’s recall for complex, distributed vulnerabilities, affecting overall performance.

Using accuracy, precision, recall, and F1 scores ensures robust conclusions, but the test set’s bias toward known fixes might overstate performance, underrepresenting real-world challenges. Alternative metrics could provide deeper insights, though the focus on F1 scores balances imbalanced data, supporting the validity of the findings within this scope.

The hybrid model’s design introduces potential biases—LSTM’s sequential focus, Conformer’s block-based structure, and UniXcoder’s pre-training may limit adaptability. Alternative architectures or hyperparameters might shift results, affecting the model’s effectiveness, though the current tuning optimizes performance within the evaluated scope.

Focusing on source code excludes binary or executable analysis, limiting the model’s applicability to compiled threats. This constraint restricts the hybrid model to Python codebases, missing vulnerabilities in other formats, which could impact its real-world utility.

The Python-only dataset restricts the model’s generalizability to other languages, such as C or Java, where vulnerability patterns may differ. While the hybrid design is language-agnostic, its current training limits portability, affecting its applicability across diverse programming environments.

5.15 Summary

This chapter introduced structural enhancements to the vulnerability detection model by integrating Conformer architectures and Large Language Models (LLMs). The hybrid LSTM-Conformer-LLM approach significantly improved detection accuracy, achieving F1 scores up to 93% for key vulnerabilities like SQL injection. Ablation studies confirmed the complementary benefits of each component, and comparisons with existing methods demonstrated superior performance. Limitations and threats to validity were discussed, highlighting areas for future research such as extension to other programming languages. The Conformer, with 12 blocks, 8 attention heads, kernel size 5, d-model 512, and d-ffn 2048, processes graph embeddings and CSEs into fused 640-dimensional inputs, augmented with sinusoidal positional encodings for order preservation. UniXcoder fine-tuning over 30 epochs at 1e-5 learning rate adapts pre-trained layers for semantic feature extraction, yielding 512-dimensional vectors that capture intent-based risks like untrusted inputs. Fusion concatenates outputs into 1152 dimensions, processed through ReLU dense and sigmoid layers for binary classification, with class weights addressing imbalances. Hyperparameter tuning, including graph dimensions and sequence lengths, balances locality and context, enhancing detection for multi-block vulnerabilities like command injection. Ablation showed full hybrid outperforming subsets, validating the tripartite synergy for robust, scalable Python code analysis.

Contributions. The novel advancements introduced in this chapter, primarily attributable to the author, encompass:

- Incorporation of a tripartite hybrid architecture that fuses LSTM for sequential dependencies, Conformer for multi-block structural patterns, and UniXCoder for deep semantic context, with detailed fusion via vector concatenation and dense layers for enhanced binary classification
- Comprehensive hyperparameter optimization for individual components, covering neuron counts, attention heads, kernel sizes, embedding dimensions, and fine-tuning rates, with tabulated results and graphical analyses for performance across configurations
- In-depth ablation study quantifying the impact of removing components or structural inputs like AST, DFG, CFG, and attention layers, alongside expanded comparisons with additional methods using averaged metrics over all vulnerabilities

6

Conclusion

This thesis presents our results on vulnerability detection by a novel hybrid model integrating LSTM, Conformer, and UniXcoder to detect vulnerabilities in Python source code, addressing the critical need for efficient, accurate software security tools. Building on the limitations of traditional sequential models, our work advances the field by combining sequential, structural, and semantic analysis, as detailed in the development and evaluation chapters. Here, we summarize our contributions, reflect on achievements, and outline their significance. Our first contribution is a comprehensive GitHub-mined dataset covering six vulnerability types—SQL injection, XSS, command injection, CSRF, remote code execution, path disclosure, and open redirect-enabling real-world training and evaluation. The hybrid model’s design is our core innovation, fusing LSTM’s sequential analysis, Conformer’s structural modeling (6 blocks, 8 attention heads, kernel size 5, d-model 512, d-ffn 2048), and UniXcoder’s semantic understanding (fine-tuned at $2e-5$ learning rate, class weight 20) into a unified feature set, optimized through hyperparameter tuning. This approach achieves superior performance, consistently outperforming the LSTM baseline across all vulnerabilities, with significant gains in both structurally complex and semantically demanding cases, as shown in performance evaluations. Comparatively, our model excels over other methods using similar data-mining approaches or databases, surpassing traditional neural networks, pre-trained models, and graph-based approaches by leveraging its integrated analysis, achieving the highest outcomes across all metrics. The ablation study confirms each component’s critical role, with performance declining when any is removed, highlighting the synergy of sequential, structural, and semantic features. Despite these achievements, limitations such as reliance on commit-based labeling, focus on known vulnerabilities, and Python-only data restrict broader applicability, as discussed in the limitations section. Future work aims to address these by expanding the dataset with synthetic data, enhancing context capture, supporting additional languages, and integrating anomaly detection, ensuring our model evolves into a more robust, versatile tool for vulnerability detection.

Appendices



Summary in English

We introduce a novel tripartite hybrid model designed to detect security vulnerabilities in Python source code by integrating Long Short-Term Memory (LSTM) networks for sequential pattern analysis, Conformers for structural feature extraction, and UniX-Coder as a Large Language Model (LLM) for semantic understanding. This approach addresses the limitations of traditional vulnerability detection methods, which often fail to capture the dynamic, context-dependent nature of Python code. By combining syntactic, structural, and semantic insights, our model targets six key vulnerabilities: SQL injection, cross-site scripting (XSS), command injection, cross-site request forgery (CSRF/XSRF), remote code execution (RCE) and path disclosure. The foundation of our work is a comprehensive dataset mined from GitHub, comprising 25,040 commits across 14,686 repositories, filtered and refined to ensure high-quality, real-world samples. Data collection involved scraping commits using security-related keywords from CVE, OWASP Top 10, and prior literature, with rigorous filtering to exclude showcase or exploit-oriented projects, resulting in balanced datasets for each vulnerability type. Our methodology begins with preprocessing, where code is tokenized, comments removed, and snippets created using overlapping focus windows (step size $n=5$, context $m=200$ characters) to preserve token boundaries and capture sufficient context. Embeddings are generated using Word2Vec, FastText, or CodeBERT, with optimal settings of 200 dimensions, min-count 10, and 100 iterations, retaining strings for semantic richness. Structural information is incorporated via Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs), encoded into 128-dimensional vectors and fused with 512-dimensional Code Sequence Embeddings (CSEs) from UniXCoder. The LSTM baseline, with 100 neurons, 128 batch size, 20% dropout, Adam optimizer, and 100 epochs, establishes sequential learning, achieving F1 scores around 85-87% on SQL injection. The Conformer, stacked with 6 blocks, 8 attention heads, kernel size 5, d-model 512, and d-ffn 2048, refines structural patterns, while UniXCoder, fine-tuned at $1e-5$ learning rate over 30 epochs with class weight 20, adds semantic depth. Fusion concatenates outputs into 1152 dimensions, processed through ReLU dense and sigmoid layers for binary classification. Hyperparameter tuning, guided by F1 scores on validation sets, balances efficiency and performance.

Empirical results show our full hybrid model achieving 91.3% average F1 across vulnerabilities, outperforming ablated versions and comparable methods like VulDeePecker (F1 82%) or GraphCodeBERT (F1 87%). Ablation studies confirm each component’s role, with drops of 2-4% when removing structural inputs or layers. Comparisons validate superiority over methods using similar data-mining or databases, highlighting our model’s integrated analysis. Limitations include commit-labeling biases, Python-only focus, and 512-token context, with threats like overfitting mitigated via dropout and early stopping. our contributions advance automated vulnerability detection, offering a scalable tool for Python security. Addressing These Issues Through the Dissertation’s Findings :

- **Mitigating Data Scarcity and Quality Issues:** We curated a GitHub dataset from 25,040 commits across 14,686 repositories, filtering for relevance and refining to six vulnerability types with samples like SQL injection (83,558 LOC), yielding robust training data and F1 scores up to 93% on test sets, outperforming sparse datasets in prior works.
- **Improving Labeling Accuracy and Automation:** Commit diffs enabled automatic labeling (pre-fix vulnerable, post-fix non-vulnerable), with ablation confirming minor F1 drops (1-2%) on edges, while class weights (optimal at 20) boosted recall by 5-7% for imbalanced types like CSRF.
- **Enhancing Context Capture Beyond Local Tokens:** Overlapping focus windows (n=5, m=200 characters) and sequence lengths (optimal at 512) captured multi-block dependencies, reducing false negatives in long-range vulnerabilities (e.g., command injection F1 92%), with graph integrations (128 dimensions) improving recall by 8-10% over sequential models.
- **Overcoming Python-Specific Limitations for Broader Applicability:** our hybrid design is language-agnostic, with embeddings like CodeBERT adaptable, and hyperparameter tuning (e.g., d-model 512) suggesting portability, as F1 variance across vulnerabilities (85-93%) indicates generalization potential for extensions to C/Java.
- **Reducing Overfitting and Ensuring Generalization:** Dropout (0.2), early stopping, and phased training prevented overfitting, with stable validation loss and ablation/comparisons confirming robustness, achieving 91.3% average F1 on unseen sets.
- **Handling Computational Efficiency and Scalability:** Batch sizes (128) and dimensions (512 for CSE, 128 for graphs) balanced demands, converging in 100 epochs on datasets like 96,041 SQL samples without degradation, enabling deployment while outperforming resource-heavy methods.

Thesis Point I. Sequential Modeling with LSTM and Code Embeddings

We demonstrated the utilization of LSTM networks and code embeddings in a sequential modeling approach to identify vulnerabilities in Python programming. We

determined that CodeBERT was the optimal solution by comparing the embeddings of Word2Vec, FastText, and CodeBERT. This enabled the LSTM model to achieve high performance metrics, such as an average F1 score of 0.90 across all vulnerability categories. The technique demonstrated successful block-level detection in real-world experiments on GitHub datasets, thereby establishing the foundation for further advancements in structural and semantic analysis. The procedure entailed the removal of remarks, the division of the code into Python-specific components, and the addition of hyperparameters, including a minimum count of 10, 200-dimensional vectors, and 100 iterations. Additionally, it maintained strings to enhance semantic richness. The data collection procedure entailed the mining of GitHub contributions that contained security-related keywords, the filtering of Python files that were less than 10,000 characters in length, and the partitioning of these files into overlapping snippets with attention windows of 5 and contexts of 200 characters. Adam optimizer, 100 epochs, 128 batch size, 20% dropout, and 100 neurons were identified as the optimal settings for LSTM tuning. CodeBERT achieved F1 scores as high as 0.83 for SQL injection. The performance breakdowns by vulnerability indicate that sequential patterns, such as hazardous concatenations, are robust, while global dependencies indicate that hybrid extensions are required.

Thesis Point II. Conformer-LLM Integration for Block-Level Vulnerability Detection

The hybrid LSTM-Conformer-LLM method significantly improved the accuracy of detection, achieving F1 scores of up to 93% for critical vulnerabilities such as SQL injection. Ablation studies confirmed the synergistic benefits of each component, and comparisons with conventional procedures demonstrated improved performance. We discussed the potential for further research, such as the expansion to other programming languages, and the hazards and limitations to validity. The Conformer is equipped with 12 blocks, 8 attention centers, a kernel size of 5, a d-model of 512, and a d-ffn of 2048. It converts graph embeddings (ASTs, CFGs, DFGs at 128 dimensions) and CSEs (512 dimensions) into fused 640-dimensional inputs, incorporating sinusoidal positional encodings to preserve the order. UniXcoder modifies pre-trained layers for semantic feature extraction using a learning rate of 1e-5 and 30 epochs of fine-tuning. This leads to 512-dimensional vectors that capture intent-based hazards, such as untrusted inputs. Binary classification is achieved by combining outputs into 1152 dimensions and processing them using ReLU dense and sigmoid layers in Fusion. Class weights rectify imbalances. Setting the graph dimensions to 128 and the sequence lengths to 512 is an example of hyperparameter tailoring, which achieves a balance between locality and context. This approach facilitates the identification of multi-block vulnerabilities, such as command injection. The tripartite synergy for resilient and scalable Python code analysis was confirmed by ablation, which demonstrated that the complete hybrid outperformed its subsets.

A.1 Summary in Hungarian

Tézisem egy újszerű hibrid MI modellt mutat be Python programokban található sebezhetőségek detektálására, amely integrálja az LSTM-et, a Conformer-t és az UniX-codert, támogatva ezzel a hatékony és pontos szoftverbiztonsági eszközök iránti kritikus igényt. A hagyományos szekvenciális modellekre építve munkám annak korlátait próbálja meghaladni a szekvenciális, strukturális és szemantikai elemzés kombinálásával. A tézisben összefoglalom személyes tudományos hozzájárulásom, valamint áttekintem az eredményeket, és felvázolom azok jelentőségét.

Első tézispontom fő eredménye egy átfogó, GitHub-ról bányászott adathalmaz, amely hét sebezhetőségi típust tartalmaz – SQL injekció, XSS, parancs injekció, CSRF, távoli kód futtatás, elérési út közzététele és nyílt átirányítás –, lehetővé téve az MI modellek betanítását és kiértékelését valós világbeli adatokon. Eredményem továbbá a különböző kódbeágyazások (word2vec, fastText, CodeBERT) hatékonyságának kiértékelése a sérülékenység előrejelzésben LSTM segítségével.

Ezen adatokra, valamint az LSTM modellel elért eredményeimre építve kidolgoztam egy hibrid modellt, amelyet a második tézispont mutat be. A hibrid modell az LSTM szekvenciális elemzését, a Conformer strukturális modellezését és az UniXcoder szemantikai megértését egyesíti. Összehasonlításképpen, modellünk kiemelkedik a hasonló adatbányászati megközelítéseket vagy adatbázisokat használó más módszerekhez képest, felülmúlja a hagyományos neurális hálózatokat, az előre betanított modelleket és a gráf-alapú megközelítéseket az integrált elemzés kihasználásával, és minden metrika szerint a legjobb eredményt éri el. Az átlagos F1 érték 91,3% a CodeBERT 85,0%-val és a GraphCodeBERT 87,0%-val szemben. Az ablációs tanulmány megerősíti az egyes komponensek kritikus szerepét, a teljesítmény csökken, ha bármelyiket eltávolítjuk, kiemelve a szekvenciális, strukturális és szemantikai jellemzők szinergiáját.

Bibliography

- [1] National institute of standards and technology. <https://www.nist.gov/>. Accessed: 2024-12-30.
- [2] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350. IEEE, 2006.
- [3] Maxat Akbanov, Vassilios G Vassilakis, and Michael D Logothetis. Wannacry ransomware: Analysis of infection, persistence, recovery prevention and propagation mechanisms. *Journal of Telecommunications and Information Technology*, (1):113–124, 2019.
- [4] Wathiq Laftah Al-Yaseen, Zulaiha Ali Othman, and Mohd Zakree Ahmad Nazri. Multi-level hybrid support vector machine and extreme learning machine based on modified k-means for intrusion detection system. *Expert Systems with Applications*, 67:296–303, 2017.
- [5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.
- [6] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pages 207–216. IEEE, 2013.
- [7] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.
- [8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
- [9] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*, 2016.
- [10] Krishna Kumar Chaturvedi, VB Sing, and Prashast Singh. Tools in mining software repositories. In *2013 13th International Conference on Computational Science and Its Applications*, pages 89–98. IEEE, 2013.

- [11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [12] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. Evaluation of chatgpt model for vulnerability detection. *arXiv preprint arXiv:2304.07232*, 2023.
- [13] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004.
- [14] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.
- [15] Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.
- [16] MITRE Corporation. Common vulnerabilities and exposures (cve). <http://cve.mitre.org/>. Accessed: 2025-09-02.
- [17] MITRE Corporation. Cwe-89: Improper neutralization of special elements used in an sql command ('sql injection'). <https://cwe.mitre.org/data/definitions/89.html>, 2019. Accessed: 2019-10-03.
- [18] MITRE Corporation. Cwe-22: Improper limitation of a pathname to a restricted directory ('path traversal'). <https://cwe.mitre.org/data/definitions/22.html>, 2025. Accessed: 2025-08-16.
- [19] MITRE Corporation. Cwe-352: Cross-site request forgery (csrf). <https://cwe.mitre.org/data/definitions/352.html>, 2025. Accessed: 2025-08-16.
- [20] MITRE Corporation. Cwe-77: Improper neutralization of special elements used in a command ('command injection'). <https://cwe.mitre.org/data/definitions/77.html>, 2025. Accessed: 2025-08-16.
- [21] Hoa Khanh Dam, Truyen Tran, John Grundy, and Aditya Ghose. Deepsoft: A vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 944–947, 2016.
- [22] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.
- [23] JackB Dennis. Data flow graphs. In *Encyclopedia of parallel computing*, pages 512–518. Springer, 2011.
- [24] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

-
- [25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [26] OWASP Foundation. Owasp foundation, the open source foundation for application security. <https://owasp.org>, 2025. Accessed: 2025-08-16.
- [27] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.
- [28] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.
- [29] GitHub. Searching commits. <https://help.github.com/en/articles/searching-commits>, 2019. Accessed: 2019-09-01.
- [30] GitHub. About github’s search rest api endpoints. <https://docs.github.com/en/rest/search/search>, 2025. Accessed: 2025-08-19.
- [31] GitHub. Github · build and ship software on a single, collaborative platform. <https://github.com/>, 2025. Accessed: 2025-08-16.
- [32] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2016.
- [33] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.
- [34] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. Conformer: Convolution-augmented transformer for speech recognition. In *Proc. Interspeech 2020*, pages 5036–5040, 2020.
- [35] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unix-coder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.
- [36] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI conference on artificial intelligence*, 2017.
- [37] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

- [38] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. Learning to repair software vulnerabilities with generative adversarial networks. *Advances in Neural Information Processing Systems*, 31, 2018.
- [39] Jeff Heaton. Ian goodfellow, yoshua bengio, and aaron courville: Deep learning: The mit press, 2016, 800 pp, isbn: 0262035618. *Genetic programming and evolvable machines*, 19(1):305–307, 2018.
- [40] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.
- [41] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012.
- [42] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. Diploma, Technische Universit"at M"unchen, 1991.
- [43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [44] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10, 2012.
- [45] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, H erve J egou, and Tomas Mikolov. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.
- [46] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. Towards a taxonomy of approaches for mining of source code repositories. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.
- [47] Mikhail V Koroteev. Bert: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943*, 2021.
- [48] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.
- [49] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005.
- [50] Kui Liu, Dongsun Kim, Tegawend  F Bissyand , Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2018.
- [51] FRANCESCO Lomio. Machine learning for software fault detection. 2022.

-
- [52] Stefan Micheelsen and Bruno Thalmann. A static analysis tool for detecting security vulnerabilities in python web applications. *Aalborg University, Aalborg University, 31st May*, 2016.
 - [53] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015.
 - [54] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530. IEEE, 2008.
 - [55] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540, 2007.
 - [56] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.
 - [57] OpenAI. Chatgpt. <https://chat.openai.com>, 2025. Accessed: 2025-08-19.
 - [58] OpenAI. Gpt-4. <https://openai.com/gpt-4>, 2025. Accessed: 2025-08-19.
 - [59] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548. IEEE, 2015.
 - [60] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.
 - [61] Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM systems journal*, 46(2):265–288, 2007.
 - [62] Bugzilla Project. Bugzilla. <https://www.bugzilla.org/>, 2025. Accessed: 2025-08-16.
 - [63] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.
 - [64] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.
 - [65] Cross-site Scripting. Owasp™ foundation. *The free and open software security community*. URL: [https://www.owasp.org/index.php/Cross-site_scripting\(XSS\)](https://www.owasp.org/index.php/Cross-site_scripting(XSS)), 2001.

- [66] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013.
- [67] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C Briand. Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 642–651. IEEE, 2013.
- [68] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.
- [69] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317, 2008.
- [70] Koustuv Sinha, Robin Jia, Dieuwke Hupkes, Joelle Pineau, Adina Williams, and Douwe Kiela. Masked language modeling and the distributional hypothesis: Order word matters pre-training for little. *arXiv preprint arXiv:2104.06644*, 2021.
- [71] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018.
- [72] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [73] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2237–2248. IEEE, 2023.
- [74] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- [75] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.
- [76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [77] David Wheeler. Flawfinder home page. *Web page: <http://www.dwheeler.com/flawfinder>*, 2006.

-
- [78] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.
 - [79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.
 - [80] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368, 2012.
 - [81] Fabian Yamaguchi, Konrad Rieck, et al. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011.
 - [82] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 52–63. IEEE, 2019.
 - [83] Zhe Yu, Christopher Theisen, Laurie Williams, and Tim Menzies. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering*, 47(11):2401–2420, 2019.
 - [84] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.
 - [85] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.
 - [86] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third international conference on software testing, verification and validation*, pages 421–428. IEEE, 2010.
 - [87] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on software engineering*, 31(6):429–445, 2005.
 - [88] Maede Zolanvari, Marcio A Teixeira, Lav Gupta, Khaled M Khan, and Raj Jain. Machine learning-based network vulnerability analysis of industrial internet of things. *IEEE internet of things journal*, 6(4):6822–6834, 2019.