

Article

Towards a Block-Level Conformer-Based Python Vulnerability Detection

Amirreza Bagheri  and Péter Hegedűs * 

Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary; bagheri@inf.u-szeged.hu

* Correspondence: hpeter@inf.u-szeged.hu

Abstract: Software vulnerabilities pose a significant threat to computer systems because they can jeopardize the integrity of both software and hardware. The existing tools for detecting vulnerabilities are inadequate. Machine learning algorithms may struggle to interpret enormous datasets because of their limited ability to understand intricate linkages within high-dimensional data. Traditional procedures, on the other hand, take a long time and require a lot of manual labor. Furthermore, earlier deep-learning approaches failed to acquire adequate feature data. Self-attention mechanisms can process information across large distances, but they do not collect structural data. This work addresses the critical problem of inadequate vulnerability detection in software systems. We propose a novel method that combines self-attention with convolutional networks to enhance the detection of software vulnerabilities by capturing both localized, position-specific features and global, content-driven interactions. Our contribution lies in the integration of these methodologies to improve the precision and F1 score of vulnerability detection systems, achieving unprecedented results on complex Python datasets. In addition, we improve the self-attention approaches by changing the denominator to address the issue of excessive attention heads creating irrelevant disturbances. We assessed the effectiveness of this strategy using six complex Python vulnerability datasets obtained from GitHub. Our rigorous study and comparison of data with previous studies resulted in the most precise outcomes and F1 score (99%) ever attained by machine learning systems.

Keywords: machine learning; deep learning; large language models; conformers; AST; DFG; CFG; CSE; LLM; vulnerability detection; Python vulnerability



Citation: Bagheri, A.; Hegedűs, P. Towards a Block-Level Conformer-Based Python Vulnerability Detection. *Software* **2024**, *3*, 310–327. <https://doi.org/10.3390/software3030016>

Academic Editor: Haipeng Cai

Received: 19 June 2024

Revised: 23 July 2024

Accepted: 29 July 2024

Published: 31 July 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The primary research problem addressed in this study is the difficulty of accurately identifying software vulnerabilities, which is crucial for preventing cybercrime and financial losses. Traditional methods are labor-intensive and often miss complex vulnerabilities. Our contribution is the development of a hybrid model that integrates self-attention mechanisms with convolutional networks, significantly improving the detection accuracy and efficiency of vulnerability identification in Python code. These vulnerabilities pose a significant threat to the security and integrity of computer systems, leading to severe consequences such as unauthorized access, data breaches, and system failures. Traditional vulnerability detection approaches, both static and dynamic, are problematic due to their labor-intensive and time-consuming nature. Automated tools for static analysis, such as Flawfinder and Findbugs, often miss complex vulnerabilities that require a thorough understanding of operational logic and system design. Given deep learning's impressive image and natural language findings, we should investigate whether it might improve vulnerability detection. However, deep learning algorithms often struggle to interpret large and complex datasets due to their limited ability to understand intricate linkages within high-dimensional data. Our findings suggest that deep learning needs to overcome additional challenges before it can effectively identify security-risky source code. To address these challenges, this research proposes a novel method that integrates self-attention mechanisms with convolutional

networks. This hybrid approach aims to enhance the detection of software vulnerabilities by capturing both localized, position-specific features and global, content-driven interactions. The widespread use of computers and the Internet has altered many human activities. Furthermore, it has raised vulnerabilities in the computing infrastructure. In 2023, the National Vulnerability Database (NVD), which is overseen by the National Institute of Standards and Technology (NIST) [1] of the United States Department of Commerce, discovered a record 225,233 vulnerabilities, emphasizing cyber security concerns. Several methodologies and strategies have enhanced vulnerability detection in professional cyber security and academic research. Traditional vulnerability discovery techniques include static and dynamic analysis. Professionals employ automated tools for static analysis of source code, such as Flawfinder [2] and Findbugs [3]. Monitoring a program's execution via dynamic analysis reveals security weaknesses. Manual procedures are arduous and prone to error, but they have advantages. Automated approaches may miss complicated vulnerabilities that need a thorough understanding of operational logic and system design. As a result of machine learning technologies, new methods using support vector machines, such as VCCFinder [4], have emerged. Ghaffarian and colleagues conducted research on software vulnerability analysis [5]. A comparison of machine-learning algorithms revealed that the random forest method performed the best in terms of accuracy and efficiency. However, even the best-performing algorithms struggle with poor feature capture when dealing with complex datasets. When dealing with complicated, large-scale industrial datasets, poor feature capture can imperil these solutions. Deep learning advancements have enabled vulnerability discovery using graph-based approaches such as FUNDED [6], Design [7], and semantic-based tactics like CodeBERT [8] and CodeT5[9]. Large language models (LLMs), such as GPT-4 [10], have significantly enhanced semantic techniques. Moreover, self-attention mechanisms, which have shown potential in processing information over long distances, lack the capability to gather structural data. This limitation hampers their effectiveness in vulnerability detection, as they cannot fully comprehend the intricate relationships within the data. Our methodology focuses on three essential elements: procedures, structural information processing, and a pre-trained machine learning model [11] to address these difficulties. Convolution and self-attention detect long- and short-range dependency in input sequences. Compared to the Transformer, this architectural approach enhances computation speed while simplifying complex network models. We tokenize code fragments to create contextually appropriate feature arrays. To reduce noise in attention processes, we modified softmax's self-attention methods. All of our code and dataset are publicly available (<https://GitHub.com/Dr-Bagheri/Block-Level-Conformer-Based-Python-Vulnerability-Detection>) accessed on 28 July 2024. Our new dataset has facilitated our examination of latest deep learning techniques, allowing us to gain fresh perspectives on potential avenues for research and the obstacles encountered in vulnerability detection using machine learning. Specifically, we investigate the following research questions:

- Research Question 1:
Does increasing the amount of training data have a positive impact on the performance of models, or have they reached their maximum potential?
- Research Question 2:
Does the design of the model have a significant influence on its effectiveness?
- Research Question 3:
Do large language models perform better in detecting vulnerabilities than the state-of-the-art models depending on code-structure features?

2. Related Work

2.1. Conventional Methodologies

The research problem identified in the related work is the inefficiency and inaccuracy of conventional vulnerability detection methods, which rely heavily on manual effort and often fail to detect complex vulnerabilities. Our contribution is the introduction of a machine-learning-based approach that leverages deep learning and large language models

to automate and enhance the accuracy of vulnerability detection. Classical vulnerability detection began with educated professionals manually creating exceptional rule bases. The rule bases built by specialists are cited in publications such as Flawfinder [2] and Vrust [12]. Scientists have developed semi-automated methods such as fuzzing, symbolic execution, taint tracking, and code similarity vulnerability detection [13–16]. These methods aim to reduce rule-based construction costs. Although these methods are effective, they necessitate a significant amount of labor and lack complete automation.

2.2. Machine-Learning-Based Approaches

Machine learning methods automate vulnerability discovery, reducing manual work. Popular algorithms include random forests, support vector machines, MLPs, and logistic regression. Al-Yaseen et al. developed a hybrid intrusion detection model with extreme learning and support vector machines [17] at many levels. They also used a customized k-means method to optimize the training datasets. Ghaffarian et al. [5] discovered that the random forest technique outperformed other machine learning vulnerability detection algorithms. Lomio et al. [18] conducted an empirical study that investigated a variety of machine-learning algorithms for vulnerability detection. They decided that the current measurements are inadequate. This study also implies that ensemble-based classifiers are more effective. Zolanvari et al. [19] investigated the effectiveness of machine learning in detecting vulnerabilities in IoT environments. Although successful, these strategies fail in complex situations.

2.3. Deep-Learning-Based Approaches

Deep learning techniques employ complicated neural networks to assist vulnerability detection algorithms in solving difficult challenges. The VulDeePecker model, developed by Li et al. [20], trains neural networks with flat language source code sequences. This weakens the code's complex semantics. Therefore, researchers have studied graph or tree topologies like the control flow graph (CFG) and abstract syntax tree (AST) for training purposes. In 2017, Allamanis et al. [21] suggested strategies for expanding gated graph neural networks and transforming source code into graphs. Wang et al.'s study, "Empirical Deep Learning for Vulnerability Detection", proposed FUNDED [6], a graph-based vulnerability detection method. Steenhoek et al. [22] conducted an experiment to see how the size of a dataset affected the performance of common models. They studied common datasets. Hin et al.'s study LineVD [23] investigated the application of graph neural networks to identify vulnerabilities. Addressing statement- and function-level data disparities improves function code prediction while avoiding vulnerabilities. Our research builds on previous results to improve vulnerability detection. We also categorize them in Table 1 for the purpose of illustration.

Code2Vec [24]: Code2Vec is a model that learns distributed representations of code by treating programs as collections of paths in abstract syntax trees (ASTs). Despite its focus on representing code snippets as continuous vectors (code embeddings), it lays the groundwork for models that could potentially adapt for tasks such as vulnerability detection through training on suitable datasets of raw Python source code. CodeBERT [8]: As previously mentioned, we train CodeBERT, a bimodal model, on both natural language and programming language data, including Python. It showcases the effective understanding and processing of code through direct training on raw source code and natural language annotations. GraphCodeBERT [25]: GraphCodeBERT extends the CodeBERT model by incorporating data flow into code representation, enabling it to understand the semantic relationships within code more accurately. It uses raw source code in conjunction with its data flow graph representation for pre-training, demonstrating another angle for effectively leveraging raw Python code. CuBERT [26]: CuBERT is another approach that pre-trains a deep learning model on a large corpus of source code, including Python. Its design aims to comprehend the context and semantics of code at a detailed level, thereby serving as another example of direct training on raw source code. Py150 [27]: Although not a

method itself, the Py150 dataset is a collection of 150,000 Python source files used for training machine learning models on tasks related to Python code, such as code completion, prediction, and potentially vulnerability detection. References to models trained on this dataset illustrate the practical application of raw source code training.

Table 1. Different approaches.

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.	-	-	-	-	✓

2.4. Large Language Model-Based Approaches

This category reflects a focused subset of methods based on deep learning. Researchers often use pre-trained large language models and fine-tune them for specific topics. Llama [28], CodeX [29], and GPT-4 [10] are a few examples. Pearce et al. [30] and Cheshkov et al. [31] both looked at how well large language models worked in this area. These models' high parameter requirements drive up training costs, even though they do not provide any discernible advantages over generic deep learning models.

3. Background

3.1. Control and Data Flow Graphs

Control flow graphs as well as data flow graphs are used as inputs to our computational model, providing critical structural information. We base the methodical analysis of injected code snippets on graph-based frameworks. In the context of code analysis, graphs offer several advantages. First of all, they capture the structural qualities that are inherent in code, which makes it easier to comprehend data dependencies and control flow structures. This comprehensive perspective enables the identification of intricate relationships inside the software. Secondly, by focusing on the important aspects of the code and ignoring the rest, graphs allow abstraction. This abstraction significantly reduces its analytical complexity. Finally, graph-based formulations provide an organized input that is useful for predictive modeling since they are inherently compatible with other computational approaches and machine learning paradigms.

3.1.1. Abstract Syntax Trees

An abstract syntax tree (AST) represents the abstract syntactic structure of the source code. It has been widely used in software engineering tools and programming languages. Nodes in an AST correspond to constructs or symbols in the source code. ASTs are abstract and lack some characteristics, including punctuation and delimiters, as compared to plain source code. On the other hand, we can use ASTs to characterize the syntactic and lexical elements of source code, like the control flow structure of the while statement and the method name readText. A number of researchers employ ASTs directly in token-based techniques for source code differencing, program repair, and source code search. These techniques are only partially able to capture source code syntactical information due to token-based approaches' limitations.

3.1.2. Control Flow Graphs

Numerous program analysis methods, including data flow and data dependence analysis, depend on the presence of a control flow graph (CFG), a basic data structure that shows every possible path a program may take via its control flow during execution. It is feasible to carry out program verification, identify software problems, and create test cases by looking at and evaluating CFGs.

3.1.3. Data Flow Graphs

The two types of nodes in a data flow graph, known as links and actors, are bipartite-directed graphs. To put it simply, when there are tokens in every input arc and none in any output arc, nodes are ready for execution. An enabled node consumes tokens on input arcs and produces tokens on output arcs. Arcs can be data arcs or control arcs. Tokens in data arcs have the types integer, real, or character; tokens in control arcs have the type Boolean. Control tokens denote sequence control, and the input control arcs only activate particular actors when the appropriate control values appear.

3.1.4. Code Sequence Embedding

The process of creating word embeddings using pre-trained models is the foundation of code sequence embedding (CSE). A pre-trained model converts each token into a feature vector, also known as a contextual token representation. Unlike conventional word-embedding approaches, which require extensive training, CSE uses pre-training strategies to reduce the chance of overfitting when there is a lack of bias in the training data.

3.2. Transformer

The Transformer model architecture was made by Vaswani et al. to get around the problems that convolutional and recurrent neural networks have with capturing long-range dependencies [32]. It excels at sequential or organized input analysis tasks, such as machine translation and natural language understanding. The Transformer's self-attention mechanism enables the model to concentrate on distinct segments of the input sequence during prediction. This method ensures excellent parallelizability and decreases training time by helping the model understand local and global situations. An encoder-decoder design employs a multitude of layers of feed-forward and self-attention neural networks.

3.3. Conformer

The conformer architecture, designed by Anmol Gulati et al. [11], improves the Transformer model by resolving its constraints. The Transformer captures global contexts well, but its computational cost quadratically grows, making extended sequence processing less efficient. To solve this problem, the conformer successfully combines convolutional and self-attention methods. An innovative conformer module, the convolutional feed-forward module, efficiently captures local dependencies. The conformer consistently performs well in long sequences due to its architectural characteristics. The conformer also uses self-attention to capture global situations well.

3.4. Large Language Models

Over the last ten years, large language models have become ground-breaking tools in the field of natural language processing. Large parameter counts, often ranging from hundreds of millions to billions, distinguish these models. This allows them to comprehend intricate linguistic structures and generate text that is relevant to the context. Notable instances are OpenAI's GPT series, Google's BERT, and Microsoft's T5. There is a lot of promise for these models in the field of code analysis. When provided with code samples, they can assist developers with various tasks such as code auto-completion, problem spotting, and refactoring advice. Their innate capacity to comprehend programming languages' context and syntax makes them priceless resources in software development projects.

4. Approach

The research problem we address in our approach is the challenge of effectively preprocessing and analyzing large datasets of raw source code to detect vulnerabilities. Our contribution is a comprehensive model that includes data mining, preprocessing, structural analysis, and the use of a conformer mechanism to improve the detection of vulnerabilities. This approach ensures that both structural and semantic features are captured, enhancing the model's accuracy. The method begins by extracting structural information using open-source technology. A specific big language model for code embedding creates a semantic feature matrix. We then use the conformer technique to extract vulnerability characteristics from both the structural and semantic data. We have implemented modifications to the self-attention processes in the conformer to address the issue of unnecessary attention heads contributing to irrelevant noise. Ultimately, we employ a multi-layer perceptron to ascertain the presence or absence of vulnerabilities. Figure 1 displays the entire model architecture, with each step explained in the subsequent sections.

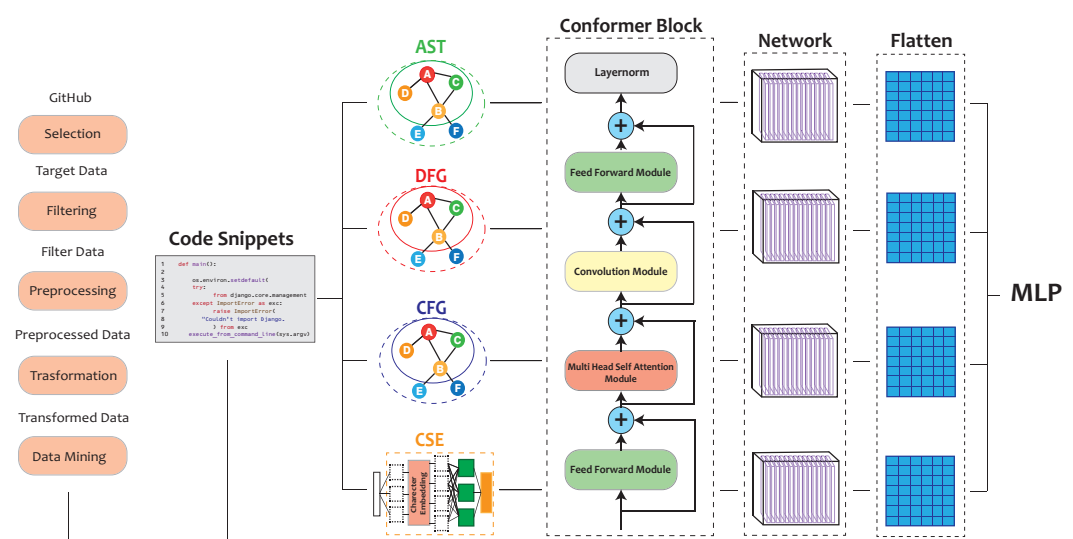


Figure 1. Design of the model.

4.1. Dataset

After collecting and filtering a large number of commits that fix vulnerabilities, we created separate datasets for each type of vulnerability. Table 2 provides a summary of the basic information about the collected vulnerabilities, including the number of repositories and commits that make up the dataset, the number of modified files that contain known vulnerabilities, the number of lines of code (LOC), the number of distinct functions they contain, and the total number of characters. The dataset is available in our repository on GitHub.

Table 2. Vulnerability dataset.

Vulnerability	Repository	Commits	Files	Functions	LOC
SQL Injection	632	871	1225	9822	203,527
XSS	122	159	157	1142	68,916
Command injection	428	824	952	6762	124,032
XSRF	211	219	584	8413	102,198
Remote code exe	272	158	686	5198	60,591
Path disclosure	574	413	732	8596	92,324

4.1.1. Data Source

Our strategy aims to utilize a vast dataset of real-world source code to train a model that is applicable to any code, not limited to a single project. We compiled the entire dataset

from publicly available GitHub projects for numerous reasons. First, because GitHub is the world's largest repository of source code, the amount of meaningful data accessible is unlikely to be insufficient for this application. Second, unlike synthetic code bases, nearly all GitHub projects contain "natural" source code, meaning they are actual projects that have been used in the field. Third, the data is open, making it easier to re-examine and reproduce the work, which is difficult in studies that focus on proprietary code, for example. Because GitHub is primarily a version control system, it is centered on commits, and as Zhou et al. [33] explain, it is possible to detect vulnerabilities by looking at commits. Patches are commits that address a defect or vulnerability and consist of two versions: one buggy and one updated and (hopefully) correct. We can discover vulnerable code patterns by evaluating the differences between the old and new versions.

4.1.2. Labeling

The data are tagged using information from the commit context, similar to Li et al. [34]. We can label the altered or deleted bits of code in such a commit as vulnerable, and label the version after the fix, along with all the data surrounding the affected component, as (potentially) not vulnerable. Of course, there are instances where a repair fails to address a problem, when multiple vulnerabilities coexist, or when a new vulnerability emerges. This strategy ignores everything because the key goal is simple automation without the need for human expert oversight. Unlike Li et al., this work does not include a post-processing manual check, as it would be difficult given the size of the dataset. Furthermore, everything labeled "not vulnerable" should be regarded as "at least not demonstrated to be vulnerable". The research problem in the implementation section is ensuring the accuracy and reliability of the data used to train our model. Our contribution is the meticulous verification process we employed to label the data as vulnerable or not vulnerable. This rigorous approach ensures that our model is trained on high-quality data, leading to more accurate and reliable vulnerability detection. By addressing potential data inaccuracies, we enhance the overall robustness and effectiveness of our model. However, because there are altered or deleted commits, it is highly likely that our data contains errors. Furthermore, our script also scrutinized the comments, a step that was deemed unnecessary as the majority of them were accurate. To obtain more comprehensive information on our whole procedure for labeling and filtering, please refer to our previous publication [35].

4.1.3. Transformation

Morrison et al. [36] claim that binary predictions and full file analysis provide limited insights. Developers know which files are vulnerable to security flaws. If possible, developers want to use a more precise approach at the line or instruction level. Dam et al. [37] demonstrate the existence of files with similar metrics, structure, and tokens, one clean and the other susceptible, with the same metrics. Instead of top-down file analysis, investigating small code snippets may be more promising. Our method meticulously examines every code token and its environment. Only in this way can we determine the location of the vulnerability.

4.1.4. Preprocessing the Data

Tokens at the source-code level in languages like Python include identifiers, keywords, separators, operators, literals, and comments. While some researchers omit separators and operators, others remove a large number of tokens and keep only API nodes or function calls. We remove comments from this work because they do not influence the program's behavior. Even if they predict vulnerability status, the model is designed to find vulnerable code, not learn this data. Otherwise, the source code remains unchanged. Hovsepyan et al. [38] take a similar strategy. No variables or literals are substituted with generic names; instead, everything is taken exactly as it appears in the code.

4.2. Structural Information

Our model's design relies heavily on structural information. It generates three types of graphs: abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs). You can construct both control flow graphs (CFGs) and data flow graphs (DFG) using source code. The abstract syntax tree (AST) is a hierarchical framework for defining a program's abstract syntax. Each node in the AST represents a distinct syntactic construct, and the edges between nodes describe the hierarchical relationships between them. This paradigm makes it much easier to study and understand code. The CFG defines various execution paths within a program, using nodes to represent program constructions and edges to signify transitions based on branching procedures. The graph clearly illustrates the start and finish points of a program, providing a visual depiction of the program's execution sequence. The DFG effectively portrays data interaction and interdependence between operations, with a focus on variable instantiation, modification, and use. Nodes in the DFG represent variables or operations, while edges show data relationships. The combination of AST, CFG, and DFG enables a full understanding of a program's structure, logic, and data flow. This, in turn, improves our model's capacity to detect and evaluate flaws.

4.3. Code Sequence Embedding (CSE)

The fundamental principle of code sequence embedding (CSE) is to use pre-trained models to generate word embeddings. A pre-trained model is used to convert each token into a feature vector, also known as a contextual token representation. CSE, unlike traditional word-embedding methods, does not necessitate substantial preparation. Instead, it employs pre-training methods to decrease the possibility of overfitting when dealing with insufficient or biased training data. As a result, pre-training has more relevant elements. Let x_i be a specific piece of code. Equation (1) describes the obtained representation P_i :

$$M_i = \text{model}(x_i) \quad (1)$$

4.4. Conformer

Self-attention mechanisms and convolutional neural networks improve sequence modeling in the conformer architectural design. This combination allows the conformer to collect local and global dependencies sequentially, overcoming certain Transformer model limits. Content-based global interactions and position-wise local characteristics are better recognized by the conformer than traditional Transformers. Figure 2 displays the different segments of conformers.

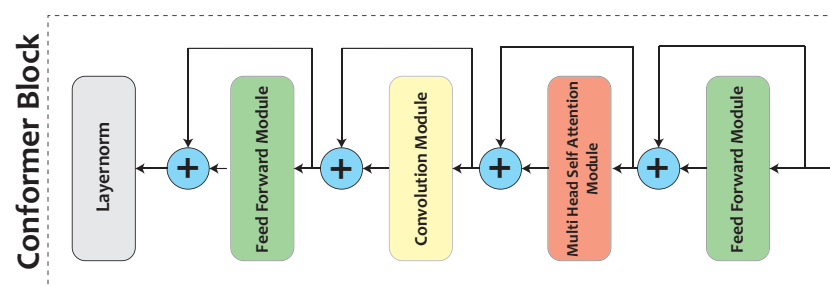


Figure 2. Conformer segments.

Self-attention records long-range relationships, while CNN module integration efficiently extracts complicated feature patterns and local context. Conformer has two feed-forward neural networks, a self-attention module, and a convolutional module. Every module is critical in processing a variety of input sequences, which improves conformer efficiency. We have also modified the standard conformer block. We put these sine-wave positional encoders together with the input matrix to do multi-head attention, and then we send the data to a fully linked layer. This change optimizes the input stage encoding of the conformer model. CNN helps the conformer architecture's convolutional module

capture sequential input local dependencies effectively. This module uses hierarchical and parallel convolutional layers to extract important data from the input sequence. We can mathematically express the output of a convolutional layer as follows:

$$\text{Conv}(x) = \text{ReLU}(\text{BatchNorm}(W * x + b)) \quad (2)$$

The convolutional layer (Conv), the rectified linear unit activation function (ReLU), the batch normalization (Batch-Norm), the convolutional kernel (W), the input sequence (x), and the bias term (b) are all parts of the equation.

Multi-head self-attention helps the self-attention module find input sequences and global interdependencies. This is achieved by focusing on multiple sequence segments at the same time. To clarify, we generate attention scores by taking the square root of the key query space dimension and normalizing the query and key vector dot products. Next, we apply a softmax function to these scores to ascertain the importance of each sequence element. These weights modulate the value vectors to produce the module's output.

$$\text{MHSA}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h) \text{WO} \quad (3)$$

The procedure effectively builds a square correlation matrix to establish correlations between differently positioned token vectors. Dot-product values make up this matrix, where each row and column represents a distinct token location. This square matrix then goes through a softmax operation on each row, producing probabilities that act as a means of merging the value vectors. The produced probability-weighted matrix then multiplies the original input vector.

$$\text{head}_i = \text{Attention}(QW_{Qi}, KW_{Ki}, VW_{Vi}) \quad (4)$$

The neural network propagates this cumulative sum to other processing layers. The multi-head attention procedure is repeated numerous times for each layer. Each attention head partitions the embedding vector and uses all its information to annotate a non-overlapping section. Softmax requires each attention head to annotate even when lacking important information, which is a drawback. Softmax works well for discrete selection problems, but not for optional annotation, particularly when the result is summative. Multi-head attention makes this issue worse since specialists contribute less than general-purpose heads.

$$\text{Attention}(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i \quad (5)$$

The model's performance is weakened as a result of the extra noise this causes. In order to fix this problem, we add 1 to the softmax equation's denominator. This stabilizes the model by guaranteeing a bounded output and a positive derivative.

$$\text{Attention}(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{1 + \sqrt{d_k}}\right) V_i \quad (6)$$

Conformer's feed-forward neural network uses non-linear adjustments to capture intricate feature correlations. It is composed of two linear layers separated by a ReLU activation function.

$$\text{FFN}(x) = \text{ReLU}(W_2(\text{ReLU}(W_1 x + b_1)) + b_2) \quad (7)$$

5. Implementation

5.1. Collecting the Dataset

5.1.1. Scraping GitHub

Creating a large dataset of GitHub security commits is the first step. To capture a wide range of vulnerabilities, we need examples of various types. We accomplish this by crafting a script that probes the GitHub API for commits related to security. We carefully design the

script, using an API token for authentication, to circumvent the search API's 1000-entry restriction and programming language filtering. We must manually sort the results to eliminate unrelated programming languages and configuration files, and select pertinent Python code. Search terms are influenced by scholarly research, the CVE database, and the OWASP security threats list. We must manually filter the results after selecting the most useful ones. As a result, this script searches the GitHub API for contributions with security-related search terms and filters out irrelevant content, such as code written in a different programming language or config files. The repository-based script uses an API token for authentication. Initially, we employed a vast array of security keywords. Terms are derived from earlier research [35], the CVE database, and the OWASP Foundation's security risk list [39]. Table 2 shows the number of downloaded repositories and files for each vulnerability to be filtered. The preprocessed dataset is also available in our repository.

5.1.2. Filtering the Results

After obtaining initial data, we filter out projects that, while rich in vulnerabilities, do not fit our code repair study. This contains efforts that demonstrate or exploit vulnerabilities rather than solve them. This filtering mechanism excludes such cases that focus on security-fixing commits. This stage also involves evaluating commit diff files to identify relevant code changes and provide context for vulnerability remedies. The objective was to discover programs that display security weaknesses, exploits, or tools for combating or preventing exploits. Since their works intentionally inject vulnerabilities into the software, they often provide useful instances of vulnerabilities but not contributions to fix them. Since the goal is to learn about vulnerable code in real-world projects where developers make valid errors, they contradict the work's methodological assumptions. Filtering such projects is an effort. The GNU diff model represents commit modifications as a diff file, and GitHub uses similar representations. It contains meta-data such as the filename, the change line number, the modified lines, and three lines of code before and after the modification. Many code differences generated by the previous process enable the reproduction of critical lines of code before and after alteration. The GitHub diff includes the updated lines, as well as three lines before and after each change, so there is little context for the change.

5.1.3. Processing the Data

Processing entails equally splitting data into vulnerable and non-vulnerable parts before labeling. This requires a sophisticated method of processing source code into blocks and deleting comments, which do not increase vulnerability. To preserve code grammar and context, a focused window iterates over the source code to prevent token segmentation. This segmentation results in overlapping sections of code, categorized as either vulnerable or clean based on their vulnerability patterns. We adjust these settings by experimenting with block sizes and focus window progression to capture vulnerability context in manageable code snippet lengths. We must treat susceptible and non-vulnerable components equally until labeling to ensure accurate data analysis. We classified equal bits of data as vulnerable if they overlapped with a problematic code portion, and as clean otherwise. We have simplified the method of dividing source code into chunks. The technique is shown in Figure 3. Initially, comments are filtered out of code, as proposed by Hovsepyan et al. [38] and others, as they are unlikely to change file vulnerability. In n steps, a small focus window iterates across the source code.

Figure 3 displays multiple blue focus window positions. The focus window always starts and terminates at a colon, bracket, or whitespace to avoid breaking tokens in half. This focus window's context is around m by m , starting and ending at code token borders. If the focus window is at the beginning of the file, the context will typically be behind it, and if it is in the middle, it will span a sample equally before and after it. Thus, many blocks overlap. If the entire block contains partially vulnerable code, it is vulnerable; otherwise, it is clean. This identifies vulnerable code fragments. The optimal values for n and m will be determined through experimentation. Liu et al. [34] suggest that a

modest 10 lines of code can capture crucial vulnerability context. The scraping and filtering method, better data processing, and specialized dataset compilation offer a high-quality, representative dataset for training a deep learning model for Python vulnerability detection. From sophisticated GitHub scraping and stringent filtering to subtle code block analysis, this entire procedure gives the machine learning model a solid foundation for identifying real-world vulnerabilities.

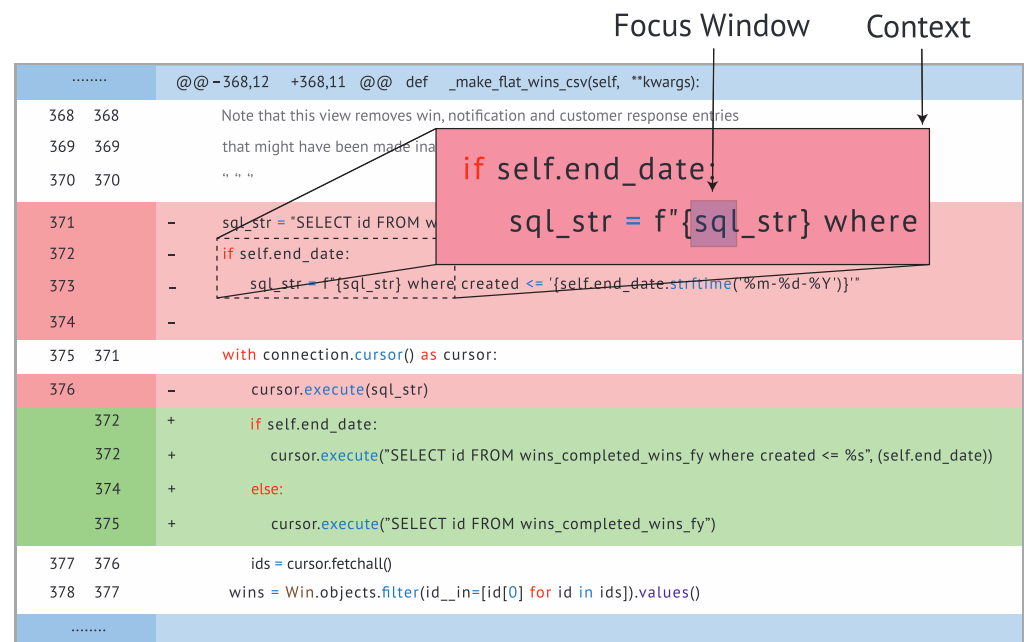


Figure 3. Processing the data from code snippet.

This integrated explanation, rich with technical detail and methodological clarity, aims to provide a thorough understanding of the complexities and nuances involved in gathering and preparing data for the critical task of vulnerability detection within Python codebases.

5.2. Building Graphs

We use tree-sitter to construct an abstract syntax tree (AST) from source code snippets. We use tree-sitter's Python language parser to evaluate source code and generate abstract syntax tree (AST) objects. We then convert these abstract syntax trees (ASTs) into Graphviz dots. We convert the dot representations into adjacency matrices by removing edges. Parsing code using the tree-sitter tool starts with CFG and DFG. Traversing AST nodes according to specified criteria yields a CFG with program execution statements or a DFG with variable declarations and updates.

5.3. Building CSE

Transformer imports UniXcoder, a cross-modal programming language pre-training model. Our experiments produced imprecise word embeddings due to tokenization issues in the original code. Our dataset's function name is mistokenized by UniXcoder. To solve this, we used the Natural Language Toolkit for tokenization. To preserve particular terms, UniXcoder uses our customized vocabulary during tokenization. UniXcoder then builds matrix code source embeddings from the source code.

5.4. Network Implementation

5.4.1. Multi-Head Self-Attention

Our multihead self-attention layer has three parameters: the number of heads, the embedding dimension, and an optional dropout rate. The layer has several components: the attention mechanism calculates attention scores, while the query-dense, key-dense, and

value-dense components conduct linear transformations on the input. The `call` function combines the heads from the `separate-heads` function, modifying and rearranging input for the desired result.

5.4.2. Sinusoidal Position Encoding

Our model includes sinusoidal position encoding and a sinusoidal position embedding layer. This layer calculates sinusoidal position embeddings from produced position IDs and indices to process input sequences. The embeddings multiply by the input sequence.

5.5. LLM

We obtained a dataset from the Common Weakness Enumeration Specification to improve LLM's abilities in detecting a wider range of vulnerabilities. This dataset includes six unique categories of Common Weakness Enumeration (CWE) vulnerabilities. To be more precise, we used the CWE instances listed in the National Vulnerability Database and created our dataset using the specific information provided by the database, which we obtained from GitHub. For each example, we divided the dataset into four components: a code sample, a vulnerability indicator, the programming language used, and a contextual description. Afterwards, we extracted code fragments that had vulnerabilities and prepared them for LLM training. The LLM specifically necessitates a particular input format known as the Alpaca format. You must adhere to this requirement.

6. Evaluation

6.1. Experimental Setup

We first outline the pre-experimentation steps, which included setting up the environment and selecting the model training parameters, prior to the experiments. Next, we give a detailed explanation of the datasets used, paying particular attention to the two datasets that we curated. We can pre-train a large language model (LLM) using one dataset, and train our model using the other. We then go on to provide comparison tests against industry standards, proving our model's effectiveness in flaw identification. We also undertake a decomposition analysis to look at the connection between the main influencing elements and the model's performance. To wrap up our assessment, we offer a comprehensive case study that delves into the effectiveness of our model in detecting software flaws as well as possible practical uses.

6.2. Performance Metrics

Here is a quick explanation of the F1-score and accuracy performance measures that we used to evaluate the model's performance.

- **Accuracy:** This metric calculates the percentage of samples that have been successfully identified relative to all samples. It can be calculated mathematically as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

- **F1-Score :** The F1-score provides a balance between precision and recall by taking the harmonic mean of the two criteria. When there is an imbalance in the classes, it is especially helpful. It is the harmonic mean of memory and precision in mathematics:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Precision:** The percentage of affirmative identifications that were in fact accurate is measured by precision. It centers on the false positive rate. In terms of math:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

- Recall: The percentage of true positive cases that were accurately detected is measured by recall. The rate of false negatives is the main focus. In terms of math:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

6.3. Environment Configuration

We used an HPE called Komondor to run the experiments. The Komondor AI [40] partition consists of 4 nodes (HPE Apollo 6500 Gen10Plus), each with 2 64-core AMD EPYCTTM 7763(Milan) CPUs and 512 GB RAM, 8 NVIDIA A100 GPUs (32 GPUs in total), 2×200 Gb/s Slingshot interconnect Rpeak = 0.6+ PF, in a high-performance workstation that serves as the computational environment for this investigation. In addition, the system has four 32-bit NVIDIA A100 Titan Core GPUs, each with 40 GB of VRAM and 512 GB of RAM. The implementation utilizes TensorFlow v2.7.0 and Keras v2.7.0 software. One of the 12 conformer blocks, which constitute the conformer encoder in the used neural network design, is utilized. Each block consists of eight attention heads and a fully connected network. We set up the Adam optimizer with a learning rate of 1–5 and a batch size of 64 to perform the optimization. We perform experiments on the datasets over 50 epochs.

6.4. Experimental Results

The research problem in the evaluation section is the need to validate the effectiveness of our proposed model in real-world scenarios. Our contribution is the rigorous evaluation of our model using various performance metrics and comparison with existing methods. The results in Tables 3–6 demonstrate that our model significantly outperforms others in terms of accuracy and F1 score, confirming its effectiveness in detecting vulnerabilities. To ensure a fair comparison, we used the same dataset for other models. However, since many of the best methods use different programming languages, we employed different approaches for testing. Firstly, we evaluated our model across different vulnerability categories using our dataset. Secondly, we compared it against alternative approaches that utilized the same data-mining and collection method but were implemented in different languages, as shown in Table 4. This allowed us to collect source codes in languages other than Python. Thirdly, we tested our model using the same dataset, but with base models that performed similar tasks in Python. Finally, we conducted tests by removing certain design components to gain further insights into our model.

The research problem in the evaluation section is the need to validate the effectiveness of our proposed model in real-world scenarios. Our contribution is the rigorous evaluation of our model using various performance metrics and comparison with existing methods. The results demonstrate that our model significantly outperforms others in terms of accuracy and F1 score, confirming its effectiveness in detecting vulnerabilities, according to Table 3. However, given that this is our dataset and our own design, it may contain some flaws. Therefore, we need to conduct a more thorough examination. Firstly, we manually inspect the entire dataset to ensure that we are not training the model on any incorrect information and to verify that the labels for each snippet are correctly assigned. We then evaluate our data-mining method using different models to determine if we can enhance their F1 scores and assess the impact it has on them.

Table 3. Vuldetective results for each vulnerability category.

Vulnerability	Accuracy	Precision	Recall	F1
SQL Injection	99.33%	97.82%	99.62%	98.73%
XSS	99.14%	97.51%	99.48%	98.48%
Command injection	99.21%	97.57%	99.52%	98.55%
XSRF	99.53%	97.69%	99.51%	98.72%
Remote code execution	99.20%	97.72%	99.52%	98.61%
Path disclosure	99.34%	97.89%	99.57%	98.72%

Table 4. Vuldetective comparison to other methods with same data-mining method.

Vunlnerability	Accuracy	Precision	Recall	F1
CNN	58.19%	38.2%	38.0%	38.12%
CodeBERT	90.83%	83.91%	83.83%	83.87%
SELFATT	84.01%	62.32%	62.03%	62.13%
Devign	82.50%	53.53%	53.12%	53.33%
VulDeePecker	80.70%	89.44%	89.24%	89.32%
FUNDED	88.89%	91.04%	90.74%	90.87%
DeepVulSeeker	90.80%	80.75%	80.42%	80.52%
Vuldetective	99.80%	98.29%	99.64%	98.12%

Table 5. Vuldetective comparison to other methods with same database.

Vunlnerability	Accuracy	Precision	Recall	F1
Code2Vec	43.12%	45.33%	43.21%	44.04%
CodeBERT	56.78%	58.65%	59.66%	57.32%
GraphCodeBERT	51.87%	48.56%	50.65%	49.54%
CuBERT	67.23%	65.98%	66.65%	64.99%
Vuldetective	99.80%	98.29%	99.64%	98.12%

Table 6. Ablation study.

Vunlnerability	Accuracy	Precision	Recall	F1
Vuldetective	99.80%	98.29%	99.64%	98.12%
without AST	62.32%	70.53%	70.02%	70.23%
without DFG	62.52%	67.31%	67.02%	67.13%
without CFG	63.33%	65.12%	64.81%	64.92%
without Conformer	60.52%	64.07%	63.59%	63.76%
without Attention L	61.12%	67.23%	66.81%	66.99%
without LLM	52.30%	64.75%	63.42%	63.81%

6.5. Comparison without Same Dataset

During the initial checking phase, we run a test with the same data-mining approach but not the same dataset. This is because the models run on multiple programming languages and require raw source code in those languages. We meticulously follow the processes outlined in their repository and articles, and then create a dataset in their coding language using the same data collection process we use for our own data. As a result, we are unable to use our Python dataset and must replace it with another. However, we may still assess the data-mining approach and compare the outcomes to obtain insights. The addition of structural information greatly improves our approach's performance in identifying vulnerabilities. When we compare Devign's performance with that of CNN, a conventional network, we find significant improvements in F1 scores and accuracy across all datasets.

This illustrates the effectiveness of using structural information to learn local code properties. Furthermore, FUNDED performs better than CodeBERT, suggesting that structure information has a beneficial impact on vulnerability discovery. Pre-training and attention processes have clear benefits. Compared to VulDeePecker, SELFATT performs better in terms of accuracy on the datasets. Additionally, CodeBERT, a pre-trained model, routinely outperforms SELFATT in a variety of metrics, highlighting the advantages of using pre-trained representations. An important finding is that Transformer-based models greatly benefit from the conformer block. It is clear that our model constantly performs better in terms of accuracy when compared to other Transformer-based models like CodeBERT, SELFATT, and Deep-VulSeeker. The conformer block's unique architecture combines self-attention mechanisms and deformable convolutions to better capture complex structural information in code. This is what makes it the best.

Our findings suggest that, whereas LLMs have made tremendous strides in many natural language processing applications, models with fewer parameter sets, such as DeepVulSeeker and our model, are superior for vulnerability identification. Because these smaller models use fewer resources, they require less hardware, which makes deployment simpler.

6.6. Comparison with Same Dataset

For methods that utilize raw Python source code directly to train models for various tasks, including vulnerability detection or code understanding, here are some notable methodologies, focusing particularly on their names and key references. These approaches leverage raw source code, allowing models to learn from the syntax and semantics inherent in the code itself. To improve the hypothetical performance comparison shown in Tables 3 and 4, it is important to stress an important part of our experimental protocol: the same Python source code database was used for all models, including ours and other important methods like Code2Vec, CodeBERT, GraphCodeBERT, and CuBERT. This consistency in the testing environment ensures a fair and direct comparison across all models, emphasizing our model's superior performance not as a function of database variance but rather as a result of its architectural and methodological advancements. The use of the same Python source code database for evaluating each model's performance not only reinforces the objectivity of our comparison, but also highlights several key attributes of our model that contribute to its improved performance.

6.7. Ablation Study

To comprehensively comprehend the impact of each individual module in our proposed paradigm, we performed an ablation study. The research focused on analyzing three key components: the conformer module, the Attention-modified layer, and an LLM. The results are concisely shown in Table 6. The comprehensive integration of all components yields the most optimal outcomes, indicating the individual significance of each part. We have not elaborated on data mining, a crucial aspect of this study, here. The process involves constructing three graphs and an embedding layer, followed by the utilization of the conformer layer and attention layer. Finally, the design is completed by selecting an appropriate training process. This collective evidence underscores the importance and efficacy of each component in achieving the final outcome.

6.7.1. Structural Information

We performed ablation tests on our model to gain a deeper understanding of the effects of different structural components. The table demonstrates that the presence of structural information has a significant influence on enhancing performance. Eliminating any of the graphs (AST, CFG, or DFG) led to consistent decreases in accuracy and F1-score. This highlights the importance of maintaining the structural integrity of the model. The ablation analysis we conducted reveals that every module in our model plays a substantial role in the overall performance. Eliminating a solitary element substantially reduced the ACC and F1-scores for all tasks, underscoring the significance of their interdependent relationship.

6.7.2. Conformer

The conformer module plays a vital role in improving both self-attention and convolution layers to achieve superior feature extraction and representation. Upon removing this module, there was a significant decrease in performance, with the F1-score dropping from 98% to 63%. The significant decrease, particularly in the F1-score, highlights the crucial significance of the conformer module.

6.7.3. Attention-Modified Layer

By incorporating our attention-modified layer, we improve the model's capacity to prioritize important features in the input data. Upon ablating this layer, there was a decrease in accuracy (ACC) and a commensurate decrease in F1-score. While the decline in

performance was noticeable, it was not as significant as when the conformer module was removed. This suggests that the attention-modified layer, although helpful, may not be as essential as the conformer module.

6.7.4. LLM

The purpose of the LLM is to offer a systematic comprehension of the input data by interpreting it in a manner that contributes significant context to the current activity. The removal of the LLM led to a significant decline in performance. More precisely, the percentage for our dataset dropped significantly from 99% to 63%. The significant reductions emphasize the crucial function that the LLM plays in both activities.

6.8. Research Questions

RQ1. Does increasing the amount of training data have a positive impact on the performance of models, or have they reached their maximum potential? The research problem here is understanding the impact of training data volume on model performance. Our contribution is the empirical evidence showing that increasing the training data significantly enhances model performance, particularly in complex tasks such as vulnerability identification. This finding suggests that our model has not yet reached its maximum potential and that further improvements can be achieved with larger datasets.

RQ2. Does the model's design have a significant influence on its effectiveness? The research problem here is determining the impact of model design on its effectiveness. Our contribution is the demonstration through ablation studies that the design of our model, which integrates self-attention mechanisms with convolutional networks, significantly influences its performance. This underscores the importance of aligning the model with task-specific constraints for optimal outcomes. Our findings highlight the critical role of each component in achieving high accuracy and F1 scores in vulnerability detection.

RQ3. Is it preferable to utilize the latest model that depends on code-structure features, or is it better to employ large language models? The research problem here is evaluating the effectiveness of models based on code-structure features versus large language models in vulnerability detection. Our contribution is the finding that models relying on code-structure features are generally more effective for tasks like vulnerability detection in Python source code. These models can parse and understand the specific syntax and semantics of Python, providing a nuanced analysis that larger, more generalized language models might miss. This insight guides future research and development in the field of software vulnerability detection.

7. Conclusions

In conclusion, the research problem we addressed is the inadequacy of existing methods for detecting software vulnerabilities. Our contribution is the development of an innovative model that integrates self-attention mechanisms with convolutional networks, achieving superior performance in vulnerability detection. Our findings set a new standard in the field and open up opportunities for further research and integration with large language models. This study not only advances the state of the art in vulnerability detection but also provides a foundation for future improvements and applications in cybersecurity. The implications of our work are significant, offering a more reliable and efficient approach to safeguarding software systems against vulnerabilities. Our model utilizes code analysis techniques to extract features and detect vulnerabilities. We achieve this by transforming the code into four distinct structural representations, subsequently processed by conformer blocks. The experimental findings demonstrate that Vuldetective establishes a novel technological standard for identifying vulnerabilities in real-world open-source projects by applying machine learning methodologies. Furthermore, we perform both ablation studies and case studies to thoroughly investigate the complexities of the model. In the future, we expect to have a lot of opportunities to integrate extensive existing language models with

our exclusive network and collaborate with LangChain to create a complete repository of information aimed at identifying vulnerabilities.

Author Contributions: Conceptualization, A.B. and P.H.; methodology, A.B.; software, A.B.; validation, A.B., P.H.; formal analysis, A.B.; investigation, A.B.; resources, A.B.; data curation, A.B.; writing—original draft preparation, A.B.; writing—review and editing, A.B. and P.H.; supervision, P.H.; project administration, P.H.; funding acquisition, P.H.

Funding: The research was supported by the the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory and by project TKP2021-NVA-09, which has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

Informed Consent Statement: Not applicable

Data Availability Statement: <https://github.com/Dr-Bagheri/Block-Level-Conformer-Based-Python-Vulnerability-Detection/tree/main/Code/data/raw> accessed on 28 July 2024.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nist. Available online: <https://nvd.nist.gov/vuln/vulnerability-detail-pages> (accessed on 28 July 2024).
2. Ferschke, O.; Gurevych, I.; Rittberger, M. FlawFinder: A Modular System for Predicting Quality Flaws in Wikipedia. In *CLEF (Online Working Notes/Labs/Workshop)*; AAAI: Washington, DC, USA, 2012; pp. 1–10.
3. Ayewah, N.; Pugh, W.; Hovemeyer, D.; Morgenthaler, J.D.; Penix, J. Using static analysis to find bugs. *IEEE Softw.* **2008**, *25*, 22–29. [CrossRef]
4. Perl, H.; Dechand, S.; Smith, M.; Arp, D.; Yamaguchi, F.; Rieck, K.; Fahl, S.; Acar, Y. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015; pp. 426–437.
5. Ghaffarian, S.M.; Shahriari, H.R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–36. [CrossRef]
6. Wang, H.; Ye, G.; Tang, Z.; Tan, S.H.; Huang, S.; Fang, D.; Feng, Y.; Bian, L.; Wang, Z. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Trans. Inf. Forensics Secur.* **2020**, *16*, 1943–1958. [CrossRef]
7. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NIPS Proceedings—Advances in Neural Information Processing Systems 32 (NIPS 2019)*, Vancouver, Canada, 8–14 December 2019; Neural Information Processing Systems (NIPS): San Diego, CA, USA, 2019; Volume 32.
8. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
9. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv* **2021**, arXiv:2109.00859.
10. Gpt-4. Available online: <https://platform.openai.com/playground/chat?mode=chat&model=gpt-4o&models=gpt-4o> (accessed on 28 July 2024).
11. Gulati, A.; Qin, J.; Chiu, C.C.; Parmar, N.; Zhang, Y.; Yu, J.; Han, W.; Wang, S.; Zhang, Z.; Wu, Y.; et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv* **2020**, arXiv:2005.08100.
12. Cui, S.; Zhao, G.; Gao, Y.; Tavu, T.; Huang, J. VRust: Automated vulnerability detection for solana smart contracts. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 639–652.
13. Johns, M.; Pfister, S.; SAP SE. End-to-End Taint Tracking for Detection and Mitigation of iNjection Vulnerabilities in Web Applications. U.S. Patent 10,129,285, 13 November 2018.
14. Wang, D.; Jiang, B.; Chan, W.K. WANA: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection. *arXiv* **2020**, arXiv:2007.15510.
15. Dinh, S.T.; Cho, H.; Martin, K.; Oest, A.; Zeng, K.; Kapravelos, A.; Ahn, G.J.; Bao, T.; Wang, R.; Doupé, A.; et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In Proceedings of the Network and Distributed System Security Symposium, Virtual, 21–25 February 2021.
16. He, J.; Balunović, M.; Ambroladze, N.; Tsankov, P.; Vechev, M. Learning to fuzz from symbolic execution with application to smart contracts. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 531–548.
17. Al-Yaseen, W.L.; Othman, Z.A.; Nazri, M.Z.A. Multi-level hybrid support vector machine and extreme learning machine based on modified K-means for intrusion detection system. *Expert Syst. Appl.* **2017**, *67*, 296–303. [CrossRef]

18. Lomio, F.; Iannone, E.; De Lucia, A.; Palomba, F.; Lenarduzzi, V. Just-in-time software vulnerability detection: Are we there yet? *J. Syst. Softw.* **2022**, *188*, 111283. [CrossRef]
19. Zolanvari, M.; Teixeira, M.A.; Gupta, L.; Khan, K.M.; Jain, R. Machine learning-based network vulnerability analysis of industrial Internet of Things. *IEEE Internet Things J.* **2019**, *6*, 6822–6834. [CrossRef]
20. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2224–2236. [CrossRef]
21. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. *arXiv* **2017**, arXiv:1711.00740.
22. Steenhoek, B.; Rahman, M.M.; Jiles, R.; Le, W. An empirical study of deep learning models for vulnerability detection. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 14–20 May 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 2237–2248.
23. Hin, D.; Kan, A.; Chen, H.; Babar, M.A. Linevd: Statement-level vulnerability detection using graph neural networks. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; pp. 596–607.
24. Alon, A.; Zilberstein, U.; Levy, O.; Yahav, E. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
25. Guo, D.; Ren, S.; Lu, S.; Pan, J.; Zhang, C.; Feng, X.; de Rijke, M. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Virtual Event/Punta Cana, Dominican Republic, 7–11 November 2021.
26. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code. *arXiv* **2020**, arXiv:2001.00059
27. Raychev, V.; Vechev, M.; Yahav, E. Probabilistic Model for Code with Decision Trees. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2016; Volume 51, pp. 731–747.
28. Touvron, H.; Lavril, T.; Izacard, G.; Martinet, X.; Lachaux, M.A.; Lacroix, T.; Rozière, B.; Goyal, N.; Hambro, E.; Azhar, F.; et al. Llama: Open and efficient foundation language models. *arXiv* **2023**, arXiv:2302.13971.
29. Finnie-Ansley, J.; Denny, P.; Becker, B.A.; Luxton-Reilly, A.; Prather, J. The robots are coming: Exploring the implications of openai codex on introductory programming. In Proceedings of the 24th Australasian Computing Education Conference, Melbourne, VIC, Australia, 14–18 February 2022; pp. 10–19.
30. Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; Dolan-Gavitt, B. Examining zero-shot vulnerability repair with large language models. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 23–24 May 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 2339–2356.
31. Cheshkov, A.; Zadorozhny, P.; Levichev, R. Evaluation of chatgpt model for vulnerability detection. *arXiv* **2023**, arXiv:2304.07232.
32. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, 30, NIPS’17: Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017; Curran Associates Inc.: Red Hook, NY, USA, 2017.
33. Zhou, Y.; Sharma, A. Automated identification of security issues from commit messages and bug reports. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 914–919.
34. Liu, K.; Kim, D.; Bissyandé, T.F.; Yoo, S.; Traon, Y.L. Mining fix patterns for findbugs violations. *IEEE Trans. Softw. Eng.* **2018**, *47*, 165–188. [CrossRef]
35. Bagheri, A.; Hegedűs, P. A comparison of different source code representation methods for vulnerability prediction in python. In *Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, 8–11 September 2021, Proceedings 14*; Springer International Publishing: Berlin/Heidelberg, Germany, 2021; pp. 267–281.
36. Morrison, P.; Herzig, K.; Murphy, B.; Williams, L. Challenges with applying vulnerability prediction models. In Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, Urbana, IL, USA, 21–22 April 2015; p. 4.
37. Dam, H.K.; Tran, T.; Pham, T. A deep language model for software code. *arXiv* **2016**, arXiv:1608.02715.
38. Hovsepyan, A.; Scandariato, R.; Joosen, W.; Walden, J. Software vulnerability prediction using text analysis techniques. In Proceedings of the 4th International Workshop on Security Measurements and Metrics, Lund, Sweden, 21 September 2012; pp. 7–10.
39. Owasp. Available online: <https://owasp.org/www-community/attacks> (accessed on 28 July 2024).
40. Hpc. Available online: <https://docs.hpc.kifu.hu/tasks/overview.html#compute-nodes> (accessed on 28 July 2024).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.