DeepVulSeeker: A Novel Vulnerability Identification Framework via Code Graph Structure and Pre-training Mechanism

Jin Wang[†], Hui Xiao[†], Shuwen Zhong[†], Yinhao Xiao [*][†]

[†]School of Information Science, Guangdong University of Finance and Economics, Guangzhou, China.

[*]Corresponding Author

Abstract—**Software vulnerabilities can pose severe harms to a computing system. They can lead to system crash, privacy leakage, or even physical damage. Correctly identifying vulnerabilities among enormous software codes in a timely manner is so far the essential prerequisite to patch them. Unfortantely, the current vulnerability identification methods, either the classic ones or the deep-learning-based ones, have several critical drawbacks, making them unable to meet the present-day demands put forward by the software industry. To overcome the drawbacks, in this paper, we propose DeepVulSeeker, a novel fully automated vulnerability identification framework, which leverages both code graph structures and the semantic features with the help of the recently advanced Graph Representation Self-Attention and pre-training mechanisms. Our experiments show that DeepVulSeeker not only reaches an accuracy as high as 0.99 on traditional CWE datasets, but also outperforms all other exisiting methods on two highly-complicated datasets. We also testified DeepVulSeeker based on three case studies, and found that DeepVulSeeker is able to understand the implications of the vulnerbilities. We have fully implemented DeepVulSeeker and open-sourced it for future follow-up research.**

I. INTRODUCTION

Software vulnerabilities have always been a major threat in the software industry for a long time. They are still the main cause of the fragility of a computing system. It is reported by the U.S. department of commerce national institute of standards and technology (NIST) the national vulnerability database (NVD) that the number of vulnerabilities in 2021 reached 18,378, which is an unprecedented high record [1]. If they cannot be identified and fixed in a timely manner, they can pose serious potential threats to our daily life, e.g., possible financial theft from online shopping, private information leakage, or even physical damage [1]. Thus, swiftly and precisely discovering software vulnerabilities and fixing them is the key to guaranteeing the security of a computing system.

In academia, the research of vulnerability identification has been continually studied for years. In an early-stage study, vulnerability identification methods can be divided into three categories: static analysis [2], dynamic analysis [3], and primitive machine learning methods [4]. Static analysis refers to examining a software program, either with manual work or with the help of some forms of automated algorithms,

[1]https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

without actually executing it. The main drawback of static analysis is that it either requires plenty of manual efforts or consumes a lot of time due to flow analysis [5]. Different from static analysis, dynamic analysis requires actually running the program to locate bugs. Hence, dynamic analysis is not a scalable method due to its nature. Primitive machine learning methods refer to those that utilize basic machine learning schemes to identify vulnerabilities with pre-determined features. The weakness of these methods is that they have poor performance on complicated program datasets where features can vary drastically.

In recent studies, the researchers tend to address the aforementioned shortcomings by applying Deep Learning, which is fully automated and is capable of being adapted to complicated situations. Some preliminary studies have shown that Deep Learning is a feasible method to identify more vulnerabilities than aforementioned methods [6]–[8]. Yet, after we probe into these existing deep-learning-based works, we found that their performances are still under satisfaction due to several reasons. Some of the models, e.g., models based on pure Graph neural network (GNN), can only capture the structural features of a program [9]. The drawback of these models is that they neglect the rich semantic features (such as the function names and system calls) hidden within the codes. Some of the models merely extract the semantic features while ignoring the structural information presented in the codes [10]. Some of the works managed to consider the structural and semantic features, but their models were insufficiently trained for the task since they failed to realize the importance of the pre-training techniques [11].

To resolve the above drawbacks, in this paper, we propose DeepVulSeeker, a model based on a heterogeneous structural information with MetaPaths, pre-trained semantic models, and Graph Representation Self-Attention (GRSA) Encoding network. DeepVulSeeker first captures the semantic information relationship between each programmatic node and its neighbors of a piece of code with the help of a multi-layer bidirectional Transformer based pre-trained model named UniXcoder model [12]. UniXcoder can effectively encode the natural languages to program languages, providing more pertinent and precise semantic information for the training of our model. DeepVulSeeker then utilizes the semantic information to ob-

tain the structural representations of the codes through Data Flow Graph (DFG), Control Flow Graph (CFG), and Abstract Syntax Tree (AST). In the end, DeepVulSeeker combines the obtained semantic information and the structural information, and feeds them to a convolution network and a feed-forward network to predict whether a given code snippet is vulnerable. In a nutshell, DeepVulSeeker can not only learn the structural information of a program, but also can understand its semantic implications. We fully implemented DeepVulSeeker and open-sourced the implementation on Github [2]. We then evaluated DeepVulSeeker on standard Common Weakness Enumeration (CWE) datasets and two highly-complicated program datasets, i.e., QEMU and FFMPEG, with a total number of 18,519 pieces of codes, 8,125 of them are vulnerable, and the rest are bug-free. Our results show that DeepVulSeeker achieves a satisfactory performance on CWE datasets, with an accuracy as high as 0.99. As for the QEMU and FFMPEG datasets, DeepVulSeeker also has an acceptable outcome with an accuracy as high as 0.64, which, to the best of our knowledge, is higher than all existing well-known methods. We also conduct case studies demonstrating three typical cases that are not identified by other baseline approaches but were successfully identified by DeepVulSeeker. We further manually patched the vulnerabilities and fed them to DeepVulSeeker, and found out that DeepVulSeeker no longer flagged them as vulnerable, meaning that DeepVulSeeker is able to discover more complicated bugs and understand their meanings. Our contributions are threefold:

- We propose a novel vulnerability identification framework, namely DeepVulSeeker, which is capable of learning both the semantic information and the structural information of a given program, to determine if it is vulnerable. DeepVulSeeker can capture more proliferated features than other existing approaches in an automated manner. We fully implemented DeepVulSeeker and open-sourced it on GitHub.
- We conducted case studies that are absent in most of other research. The results showed that DeepVulSeeker is capable of finding more complicated vulnerabilities and better understanding their implications.
- We conducted plentiful experiments on large datasets and compared the performance with the most recent well-known methods. The results show that DeepVulSeeker outperformed most of these methods on the traditional CWE datasets and outperformed all methods on the highly-complicated program datasets, i.e., QEMU and FFMPEG.

Paper Organization. The rest of the paper is organized as follows. Section II presents the recently advanced background knowledge of our approach. Section III details the design and technical components of the DeepVulSeeker framework. Section IV demonstrates our implementation of DeepVulSeeker. Section V reports our evaluation results and case studies on DeepVulSeeker. Section VI outlines the most

related work. Section VII concludes the paper with a future research discussion.

II. BACKGROUND

In this section, we mainly demonstrate the background knowledge of some recently advanced technologies that are exploited by DeepVulSeeker.

A. Pre-trained Models

Pre-trained models are machine learning models that are trained, developed, and supplied by other developers. They are typically used to solve problems based on deep learning and are always trained on large datasets. large-scale pre-trained models (PTMs) such as BERT and GPT have recently achieved great success and have become milestones in the field of artificial intelligence (AI). The pre-training mechanism has been developed as a typical machine-learning paradigm for decades. Thanks to complex pre-training objectives and huge model parameters, large-scale PTM can effectively extract knowledge from large amounts of labeled and unlabelled data. As widely demonstrated by experimental validation and empirical analysis, the wealth of knowledge implicit in huge parameters can benefit a variety of downstream tasks by storing it in huge parameters and fine-tuning it for specific tasks. The consensus in the AI community is now to adopt PTM as the backbone for downstream tasks, rather than learning models from scratch. With the development of deep neural networks in the NLP community, the introduction of Transformers has made it possible to train very deep neural models for NLP tasks. In 2018, PTM, such as GPT and BERT, was proposed for NLP tasks, using Transformers as the architecture and targeting language model learning. From GPT and BERT, we can see that when the size of PTM becomes large, with hundreds of millions of parameters, large-scale PTMs can capture multiple sense disambiguation, lexical and syntactic structure, and factual knowledge from text. The rich linguistic knowledge of PTMs brings excellent performance to downstream NLP tasks by fine-tuning the large-scale PTMs on a large sample size. Over the past few years, large-scale PTM has performed well on both language comprehension and language generation tasks, even achieving better results than humans. There is also a consensus to fine-tune large-scale PTMs for specific AI tasks, rather than learning models from scratch.

B. Graph Neural Networks (GNN)

A graph neural network (GNN) is an optimizable transformation of all the graph properties (nodes, edges, global context) that preserve graph symmetry. This transformation means that ordering the vertex in another way does not change the connectivity of the graphs, and would not affect their physical meaning. Graph neural networks can be divided into five categories: Graph Convolution Networks, Graph Attention Networks, Graph Autoencoders, Graph Generative Networks, and Graph Spatial-temporal Networks.

1) Graph Convolution Networks (GCN): The core idea is to learn a functional map by which a node in the map can aggregate its characteristics with the characteristics of its neighbors to generate a new representation of the node. GCN methods can be divided into two categories: spectral-based and spatial-based.

- The spectral-based method introduces a filter from the perspective of graph signal processing to define graph convolution, where the graph convolution operation is interpreted as removing noise from the graph signal.
- In the space-based method, the graph convolution is represented as the aggregation of feature information from the neighborhood. When the algorithm of the graph convolution network runs at the node level, the graph pooling module can be intellect with the graph convolution layer to coarse the graph into a high-level substructure.

2) Graph Attention Networks (GAN): The attention mechanism has been widely applied to sequence-based tasks. Introducing the attention mechanism into GNN enables neural networks to focus on nodes and edges that are more relevant to tasks. It uses attention in the process of aggregation, integrates the output of multiple models, and generates random walks oriented to important targets, which can effectively improve the effectiveness of training and the accuracy of testing. This forms the graph attention network. The followings are some of the most popular uses of attention mechanisms:

- Graph Attention Network (GAT) is a space-based graph convolution network. Its attention mechanism is used to determine the weight of the node neighborhood when the feature information is aggregated.
- Gated Attention Network (GaAN) uses a multi-head attention mechanism to update the hidden state of the nodes. It uses a convolutional subnetwork to control the importance of each attention head.
- Graph Attention Model (GAM) provides a recurrent neural network model to solve the graph classification problem by adaptively accessing a sequence of important nodes to process graph information.

3) Graph Autoencoders: Graph autoencoder is a kind of graph embedding method, whose purpose is to utilize neural network structure to represent graph vertices as low dimensional vectors. The common solution is to obtain the node embedding by using the multi-layer perceptron as the encoder, in which the decoder reconstructs the neighborhood statistics of the node.

4) Graph Generative Networks: Graph Generative Network (GGN) is a kind of GNN used to generate graph data. It uses certain rules to recombine nodes and generative edges and ultimately generates target graphs with specific properties and requirements.

5) Graph Spatial-temporal Networks: The graph space-time network also captures the space-time correlation of the space-time graph. The space-time graph has a global graph structure, with the input of each node changing over time.

C. Self-Attention Mechanism

The attention mechanism refers to forcing a deep learning network to focus on a particular piece of information and reducing attention to other pieces of information during the training process. The introduction of the attention mechanism in a neural network can reasonably allocate resources, focus on the key and useful information in a considerable amount of input information, and reduce the attention to the rest of the information, which can improve the efficiency and accuracy of the processing task. Thus, it is an effective way to solve information overload.

In the experiment of our study, we utilize a self-attention mechanism, which is a variant of the attention mechanism. It differs from the attention mechanism in that the self-attention mechanism allows the modeling of the dependence relationship without considering the distance between input information and output information. It can be utilized to calculate the correlation attention mechanism between different positions in a single sentence, meaning that it can calculate the mutual influence of different words in a single sentence to achieve global correlation weight. Therefore, the self-attention mechanism is an appropriate technique for our model since it can analyze the correlation of different locations in the program for the input code snippets in order to gain a better understanding of the code structure.

III. DESIGN OF DEEPVULSEEKER

In this section, we detail the design of DeepVulSeeker. The overall structure of DeepVulSeeker is shown in Fig. 1. The workflow of DeepVulSeeker can be described as a four-step process. First, DeepVulSeeker takes the target source codes as inputs and obtains the structural information, i.e., Data Flow Graph (DFG), Control Flow Graph (CFG), and Abstract Syntax Tree (AST) graphs, represented as adjacency matrices. Second, DeepVulSeeker encodes the raw source codes into contextualized token representations, i.e., Programming Languages Sequencing (PLS), empowered by the pre-training mechanism. Third, DeepVulSeeker further applies the technique of MetaPaths to the structural information and feeds them to Graph Representation Self-Attention Encode (GRSA) network together with the PLS outcome. Last, DeepVulSeeker forwards the outcomes from the third step to a convolution network and a feedforward network, which predicts if the given target source codes are vulnerable.

A. Structural Information

Structural information is one of the most critical parts of DeepVulSeeker. In this subsection, we leverage an example shown in Fig. 2 and Fig. 3 to illustrate how the structural information is generated.

1) Abstract Syntax Tree (AST): AST is a tree representation of the abstract syntax structure of a piece of source codes. Each vertex in the tree represents a syntactic occurring of the code context, and each edge reflects the containment relationship between a pair of vertices. If we construct the AST for the example in Fig. 2, the result is a directed graph
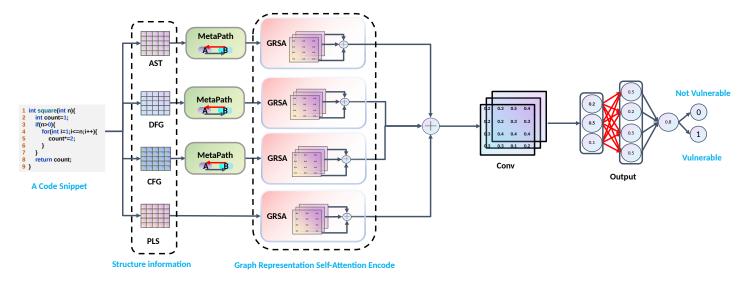
Fig. 1: Model Architecture

```
1  int square(int n){
2     int count=1;
3     if(n>0){
4        for(int i=1;i<=n;i++){
5           count*=2;
6        }
7     }
8     return count;
9  }
```

Fig. 2: A Code Example

shown in Fig. 3 in which the vertices are connected with the red edges. In this example, the token "count" in "int count=1" is connected with "int", "count", "=" and "1".

2) Control Flow Graph (CFG): CFG describes all possible control paths in a program that can be traversed. The paths to be traversed are specified by the branching statements, e.g., *if*, *while* and *do* statements. In CFG, each vertex represents a syntactic occurring, and each edge stands for a possible branch. If we construct the CFG for the example in Fig. 2, the result is a directed graph shown in Fig. 3 in which the vertices are connected with the blue edges. When the program hits the statement "if(n>0)", it decides whether to run "for(int i=1; i<=n; i++)" based on n variable. Thus, the tokens in "if(n>0)" are linked to the tokens within the "if(n>0)" statement and the tokens in "return count".

3) Data Flow Graph (DFG): DFG tracks the access of variables in a program. The access of a variable refers to both its value assignments and value modifications. In DFG, each vertex is a syntactic occurring, and each edge between a pair of vertices indicates that these two vertices are relevant to the assignments and modifications of a variable. If we construct the DFG for the example in Fig. 2, the result is a directed graph

shown in Fig. 3 in which the vertices are connected with the green edges. The variable "count", which is modified with a new value in the statement "count*=2", is declared in the statement "int count=1". Therefore, all the tokens in the statement of "count*=2" and those in "int count=1" are connected.

B. MetaPath

A MetaPath θ is a path in the form of $\theta = A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} ... \xrightarrow{R_l} A_{l+1}$, where A_i is a state and R_i is a composite relation between A_i and A_{i+1} [13]. The notation ∘ denotes the composition operator on relations. The length of a MetaPath is determined by the number of relations according to a specific model. In our model, A_i refers to a vertex from the aforementioned graphs, and R_i indicates a new edge that should be added between A_i and A_{i+1} based on a customized rule. In our design, DeepVulSeeker adopts a so-called length-2 MetaPaths rule [14], [15]. According to this rule, if there is a directed edge between two vertices, we add a reverse directed edge between them. Most subgraphs from AST, CFG and DFG are tree-like, meaning that there are very few "loops" in the graphs. Training tree-like graphs may lead to the gradient vanishing problem. Therefore, we employ length-2 MetaPaths rule to mitigate this problem by improving the completeness of the graphs.

C. Programming Languages Sequencing (PLS)

PLS is the key process for DeepVulSeeker to understand the semantic implications behind the codes. It encodes each programmatic token into a vector of features, namely contextualized token representations, based on a pre-trained model. Compared with some traditional word embedding methods which require thorough training, PLS employs pre-training techniques to avoid the model being overfitting when the size of the training data is insufficient or skewed. Therefore, pre-training techniques offer more apposite features. We denote
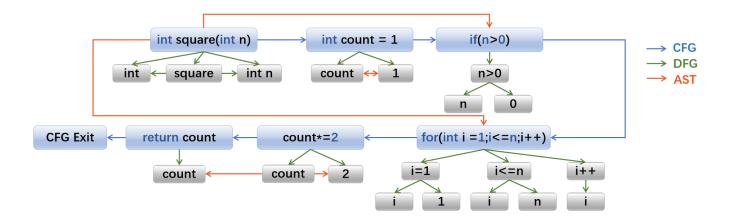
Fig. 3: Generation of Structural Information of Fig. 2

x_i as a given piece of codes, and the representations P_i are obtained by Eq. 1.

$$P_i = model(x_i), \quad (1)$$

where $model$ is the mathematical expression of a pre-trained model.

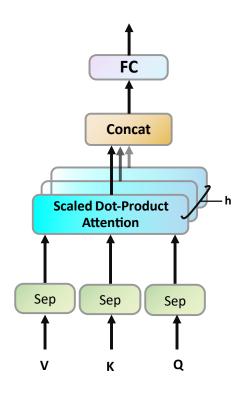### D. Graph Representation Self-Attention Encoding

Fig. 4: The Structure of GRSA

As aforementioned, we obtain three structural information (i.e., AST, CFG, and DFG), and semantic information from PLS. Our next step is to jointly combine these representations and feed them to our model for the follow-up training phase. In order to do so, we design the Graph Representation Self-Attention Encoding (GRSA) based on the multi-head self-attention mechanism [16]. GRSA is a three-step process shown in Fig. 4. First, GRSA takes three matrices, Q, K and V, as inputs, and equally separates each matrix into multiple submatrices according to the number of heads (the selections of numbers are detailed in Sec IV). The notations of Q, K and V are query, key, and value matrix respectively in the attention-based model. Since we adopt a self-attention mechanism, Q, K and V are the same. In our model, we assign CFG, DFG, AST and PLS to Q, K, and V each time. Second, GRSA retrieves the submatrices output by the first step and feeds them to a layer of scaled dot-product attention, in which the attention is calculated as Eq. 2.

$$h_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i, \quad (2)$$

where Q_i, K_i and V_i denote i-th submatrix of Q, K, and V, respectively. The notation of d_k refers to the dimension of K. In the end, GRSA concatenates all the submatrices h_i output from the scaled dot-product attention layer, and feeds them into a fully-connected network. After the process of GRSA, we manage to synthetically and effectively combine all information from the four different representations into one matrix, which contains both structural and semantic information.

IV. IMPLEMENTATION

In this section, we detail our implementation of Deep-VulSeeker shown in Fig. 1.

A. Generating Structual and Semenanic Information

As demonstrated in Sec III, the first and second steps are to generate CFG, DFG, AST and PLS information.

1) Generating CFG and DFG Adjacency Matrices:
We extracted C++ code by using tree-sitter-c [3], which is a parser generator tool and incremental parsing library that builds concrete syntax trees for source files and efficiently updates the syntax trees when editing the source files. After that, We built a Node class that records the header nodes, end node and the next node of the current node, so that later we can traversal all the nodes and their connected ones conveniently. We then used lists to store all the connected nodes, added directed edges among these nodes if they are dependent according to conditional statements, and generated the CFG adjacency matrix based on these directed edges. For example, we have a node 1, which has two child nodes 2 and 3. Then we set the matrix $M(1,2) = 1$, $M(1,3) = 1$. The generation process of the DFG adjacency matrix is similar to the one of CFG adjacency matrix. The only difference is that we added directed edges based on data flow instead of conditional statements.

2) Generating AST Adjacency Matrix: We utilized Joern [4], an open-source source code analyzer, to parse the source codes to generate Code Property Graph (CPG), which contains all the edge information as the form of $i \rightarrow j$, indicating that there is a directed edge pointing from node i to node j. We then generated the corresponding adjacency matrix according to the edge information.

3) PLS Embedding: We employed the HuggingFace [17] Transformers framework, an integrated platform for sharing and loading pre-trained models, for PLS implementation [17]. However, we found that the pre-train models from Hugging-Face incorrectly sliced the tokens for our datasets, by simply applying the slicing method for NLP to process the program codes. For example, the token `ff_insert_inpad`, which should not be split since it represents a variable in the code, was incorrectly split into tokens `ff`, `_`, `insert`, `_` and `inpad` by HuggingFace. During our practice, we discovered that incorrect slicing could greatly undermine the accuracy of our model. To resolve the problem, we utilized NLTK, an open-source natural language processing tool, to perform the token slicing task [18]. By using this tool, we were able to correctly slice the tokens from all pieces of source codes, and fed the tokens a the UniXcoder pre-trained model, a unified cross-model pre-trained model for programming languages that supports us to understand code token sequences [12]. In the end, for every piece of source codes, UniXcoder outputs a PLS embedding matrix that contains its semantic implications. Note that UniXcoder can only accept a number of 512 tokens at maximum. Thus, we trimmed the tokens from those that have more than 512 tokens to meet the requirement of UniXcoder.

B. Implementation of MetaPath and GRSA

The third step of the DeepVulSeeker process involves with the implementation of MetaPath and GRSA.

[3]https://tree-sitter.github.io/tree-sitter/
[4]https://github.com/joernio/joern

1) MetaPaths: As mentioned in Sec III, we leveraged the length-2 MetaPaths rule to realize the implementation of the MetaPaths. To do so, we wrote a Python program to add reversely-directed edges to all pairs of connected nodes. Using length-2 MetaPaths rule can mitigate the gradient vanishing problem without leading to path explosion problem as length-n MetaPaths rule does [14], [15].

2) GRSA: As mentioned in Sec III, our design of GRSA is shown as Fig. 4. We implemented the design based on Keras [19], an open-source Python-based deep learning library that is empowered by TensorFlow [20]. In the end, the outputs of GRSA layer are fed to a convolution network and a feed-forward fully-connected network, which would be trained to predict whether a given code snippet is vulnerable. We employed the cross entropy loss function for the training of our model, which is calculated as follows:

$$\mathcal{L}_{CE} = \sum_{i=0}^{N} y_i \log p_i + (1 - y_i) \log (1 - p_i), \qquad (3)$$

where y_i denotes the ground-truth labels and p_i is the probability of the label generated by the model.

V. EVALUATION

In this section, we detail the evaluation of Deep-VulSeeker. We first demonstrate the experimental setup for the evaluation. We then describe the recently state-of-the-art baseline methods and the general comparison performance between these methods and our model. We also conduct the ablation study and case study to further explore and explain the potential of our model.

A. Experimental Setup

We mainly demonstrate the performance metrics, datasets, and environment configuration used for evaluation.

1) Performance Metrics: we use two mostly used higher-is-better metrics in learning-related studies, i.e., accuracy and F1, for evaluation. We give the definitions of these two metrics in the following context. Note that we need to explain the definitions of precision and recall before explaining the one of F1.

Accuracy: The ratio of correctly labeled cases to the total number of test cases.

Precision: The ratio of correctly predicted samples to the total number of samples that are predicted to have a specific label. This metric answers questions such as "Of all the code revisions that are labeled to be vulnerability-relevant, how many are correct?". High Precision indicates a low false-positive rate.

Recall: The ratio of correctly predicted samples to the total number of test samples that belong to a class. This metric answers questions such as "Of all the vulnerable test samples, how many are labeled to be vulnerable?". High recall suggests a low false-negative rate.

F1 Score: The harmonic mean of Precision and Recall. It is an important performance indicator of a model when the

TABLE I: Dataset Information

Project	Training Set	Validation Set	Test Set	Total	Vul	Non-Vul
FFmpeg	3958	462	499	4919	2438	2481
QEMU	10903	1378	1318	13600	5687	7913
CWE-362	451	56	57	564	189	375
CWE-476	1298	162	163	1623	396	1227
CWE-754	4034	504	505	5043	1359	3684
CWE-758	1089	136	137	1362	367	995

test data have an uneven distribution of vulnerability types. F1 is calculated as follows:

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

2) Dataset: We evaluate our approach on six C/C++-language datasets that are used in many state-of-the-art studies [11], [21], [22]. The dataset contains four conventional software assurance reference datasets, namely, CWE-362, CWE-476, CWE-754, and CWE-758 [23]. These four datasets are collected from various software that have relatively simple code structures and functionalities. Thus, their vulnerabilities may be easier to be identified. The other two datasets are FFmpeg and QEMU. FFmpeg is powerful a cross-platform open-source solution to record, convert and stream video [24]. QEMU is an extremely powerful operating-system-emulator that allows running operating systems for any machine, on any supported architecture [25]. The codes from these two datasets are highly complicated. Thus, their vulnerabilities are difficult to be identified compared to those from CWE datasets. The detailed information, including the size of training sets, validation sets, test sets, numbers of vulnerabilities, and numbers of vulnerability-free are listed in Table I.

3) Environment Configuration: The hardware configuration of our experiment is trained and tested on a multi-core computing server with a 16-core 2.10 GHz Intel Xeon CPU and an NVIDIA 3090 GPU. The RAM of the server is 256 GB and the VRAM is 24GB. The software configuration of our experiment is Tensorflow v.2.7.0 and Keras v.2.7.0. running on Windows 10. For GRSA, we set head number to 4 for PLS and 2 for CFG, DFG and AST. For the layers of convonlutional and fully-connected, we use the Adam optimizer with learning rate of 1e-5 and batch size of 32. The overall training process lasts about 1.5 hours. The final trained model has more than 300,000 hyperparameters.

B. Baseline Methods

We compared the performance of our model with six baseline methods to fully demonstrate the advantage of our model.

1) VulDeePecker [26]: It treats the source code as natural languages and inputs the tokenized code into a multi-layer BILSTM. The initial embeddings of tokens are trained via Word2Vec.

2) Convolution Neural Network [27]: It treats source codes as natural languages, in which the embedding initializa-

tion is the same as that of VulDeePecker. Using Convolution Neural Network (CNN) to extract features from source codes.

3) SELFATT [28]: It transfers a piece of source codes to a token sequence and exploits the multi-head attention mechanism to learn vulnerability patterns.

4) Devign [11]: In the composite code representation, with ASTs as the backbone, it explicitly encodes the program control and data dependency at different levels into a joint graph of heterogeneous edges with each type denoting the connection regarding the corresponding representation and utilizes the gated graph neural network model with the convolution module for graph-level classification.

5) CodeBERT [29]: It is a pre-trained model for programming languages, which is a multi-programming-lingual model pre-trained on natural-language-to-programming-language (NL-PL) pairs in six different programming languages.

6) FUNDED [9]: It is a graph-learning-based approach for learning code vulnerability detection models. It extends the standard graph neural network to model multiple code relationships that are essential for modeling program structures for vulnerability detection.

C. Experimental Results

We evaluate DeepVulSeeker against six baseline methods on all six datasets shown in Table III. DeepVulSeeker outperforms the majorarity of baselines on almost all CWE datasets and outperforms all baselines on FFmpeg and QEMU. According to the experimental results, we summarize the following findings:

- **The structural information assists the model to promote the ability of vulnerability identification.** Comparing CNN and VulDeePecker, we find that the Accuracy is significantly improved in both six datasets, implying that the local characteristics learned by CNN are indeed helpful for correctly identifying vulnerabilities. Also, the fact that FUNDED outperforms CodeBERT with an average of 1.56% higher Accuracy and 6.66% F1 Score supports the finding.

- **The effect of attention and pre-training mechanisms are positive.** The Accuracy of SELFATT on FFmpeg and QEMU is 57.10% and 57.25% higher than that of VulDeePecker respectively. In addition, CodeBERT performs better than SELFATT across all metrics on two datasets, which fully reflects the advantages of the pre-trained model.
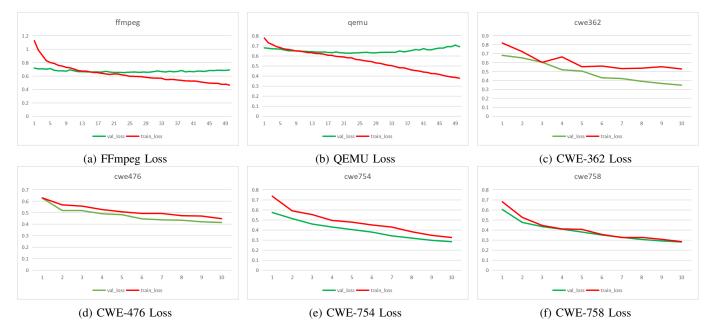
(a) FFmpeg Loss   (b) QEMU Loss   (c) CWE-362 Loss

(d) CWE-476 Loss   (e) CWE-754 Loss   (f) CWE-758 Loss

Fig. 5: Training Loss

TABLE II: Experimental Results on CWE

Method	CWE-476		CWE-758		CWE-362		CWE-754	
	ACC	F1	ACC	F1	ACC	F1	ACC	F1
CNN [27]	0.5819	0.3812	0.6731	0.2031	0.6162	0.5083	0.7371	0.343
CodeBERT [29]	0.9083	0.8387	0.8356	0.7177	0.7691	0.8695	0.5246	0.4706
SELFATT [28]	0.8401	0.6213	0.9121	0.8656	0.6627	0.0307	0.9219	0.8769
Devign [11]	0.8250	0.5333	0.8822	0.8164	0.8571	0.8148	0.9291	0.8794
VulDeepecker [26]	0.8070	0.8932	0.7887	0.8818	0.9500	0.9048	0.9574	0.9351
FUNDED [9]	0.8889	0.9087	0.9583	0.9615	0.9446	0.9521	0.9286	0.9331
DeepVulSeeker	0.9080	0.8052	0.9927	0.9859	0.9123	0.8387	0.9999	0.9999

TABLE III: Experimental Results on FFmpeg and QEMU

Method	FFmpeg		QEMU	
	ACC	F1	ACC	F1
CNN [27]	0.5493	0.5341	0.5892	0.1330
VulDeePecker [26]	0.5637	0.5702	0.5980	0.6141
SELFATT [28]	0.5710	0.5275	0.6049	0.4904
CodeBERT [29]	0.5264	0.6138	0.5354	0.7126
Devign [11]	0.5904	0.6015	0.6039	0.3244
FUNDED [9]	0.5420	0.6800	0.5970	0.7480
DeepVulSeeker	0.6354	0.6703	0.6409	0.5293

- **DeepVulSeeker may be the leading approach to date.** On the one hand, even though DeepVulSeeker slightly underperforms some baselines on some CWE datasets in regards to either the accuracy or F1, the gap is somehow trivial to tell the difference. On the other hand, DeepVulSeeker makes a breakthrough in discovering vulnerabilities in highly-complicated datasets, yielding it more properly to be adapted to real-world industry tasks.

- **Complicated datasets are more susceptible to the overfitting problem.** We show the loss changing trends during the training of our model in Fig. 5 and notice that our model has ideal loss declines in CWE dataset while less ideal loss declines in QEMU and FFmpeg data, meaning that QEMU and FFmpeg datasets are more susceptible overfitting problem. Future research can be explored on improving this problem.

D. Ablation Study

We also conduct the alation study to explore how different model components can affect the overall performance. We use FFmpeg and QEMU datasets for this study.

Structural Information. We first conduct ablation on the structural information of the model. The results are shown in Table. IV. One can see that when we remove any of the graphs, it causes drops in both accuracy and F1 scores, meaning that structural information is critical for better performance. As for the FFmpeg dataset, AST and CFG play more important roles than DFG since removing either of them results in more drastic declines than removing DFG. As for the QEMU dataset, the impacts of CFG and DFG are greater than the ones of AST.

Altogether, CFG may be the most significant graph among all three graphs, even though removing any of them may not be a wise option.

Semantic Information and Pre-trained Model. Applying semantic information and pre-training mechanism to the model is one of the biggest novel contributions produced by our approach. Thus, it is necessary to study the performance by removing them. In our model, we can simply remove the PLS process to achieve gain the results. As one can see in Table. IV, after removing the PLS, the performance of the model decreases significantly on both accuracy and F1 scores. This indicates that PLS process generates and provides crucial information to our model. If these pieces of information are missing, the performance would be not much better than a random guess.

TABLE IV: Ablation Study

Mehods	FFmpeg		QEMU	
	ACC	F1	ACC	F1
DeepVulSeeker	**0.6354**	**0.6703**	**0.6409**	**0.5293**
DeepVulSeeker w/o AST	0.5723	0.5711	0.6295	0.4816
DeepVulSeeker w/o CFG	0.5723	0.5821	0.6188	0.4405
DeepVulSeeker w/o DFG	0.5906	0.6138	0.6188	0.4056
DeepVulSeeker w/o PLS	0.5377	0.5343	0.5566	0.2422

E. Case Study

To further explore the robustness and intelligence level of DeepVulSeeker, we conduct a study on three typical vulnerable cases from the FFmpeg dataset. These three cases are correctly identified by our model but are not identified by other methods. We first analyze the vulnerability specifications of the cases. We then manually patch the vulnerability and feed them again to our model to observe whether our model would no longer flag them in order to check if our model can truly understand the meaning of the vulnerabilities in depth.

1) Case 1: In case 1 (See in Fig. 6), the code snippet is potentially vulnerable to integer overflow. In lines 2, 3, and 4, variables `lt`, `rt`, and `md` are defined as `CoefType`, which is in fact an unsigned 32-bit integer defined in FFmpeg headers. In line 4, `md` is assigned with the sum of `lt` and `rt`, which may be greater than 32-bit, causing a possible integer overflow problem. To patch this vulnerability, we simply delete line 4. We find out that DeepVulSeeker no longer flags this code snippet after the patching, implying that it can accurately understand the meaning of the integer overflow problem.

2) Case 2: In line 3 of case2 (See in Fig. 7), the program directly accesses the pointer `s` without first checking if it is a null pointer, meaning that either the memory of `s` is freed or it is not initialized. If `s` is null, then it is a dangling pointer, which can possibly lead to serious security problems such as remote code execution (RCE) by hackers. To patch this vulnerability, we simply add a statement "`if(!s) return -1;`" at the beginning of the function to check if `s` is a null pointer. After the patch, DeepVulSeeker no longer reports the

```
1  for ( i = start ; i < end ; i ++ ) {
2      CoefType lt = block -> mdct_coef [ 0 ] [ i ] ;
3      CoefType rt = block -> mdct_coef [ 1 ] [ i ] ;
4      CoefType md = lt + rt ; //Integer overflow
5      CoefType sd = lt - rt ;
6      sum [ 0 ] += lt * lt ;
7      sum [ 1 ] += rt * rt ;
8      sum [ 2 ] += md * md ;
9      sum [ 3 ] += sd * sd ;
10 }
```

Fig. 6: Case 1

program as vulnerable, implying that it is able to precisely understand the reason behind the dangling pointer problem.

```
1  static int nut_read_header ( AVFormatContext * s , AVFormatParameters * ap )
2  {
3      if(!s)return -1; //add
4      NUTContext * nut = s -> priv_data ;
5      ByteIOContext * bc = & s -> pb ;
6      int64_t pos ;
7      int inited_stream_count ;
8      nut -> avf = s ;
9      //body code
10 }
```

Fig. 7: Case 2

3) Case 3: In case3 (See in Fig. 8), a possible formatted string vulnerability can happen in line 7 when invoking the function `snprintf`. The problem with this code snippet is that `snprintf` directly takes the externally-controlled format string `s->path` without regulating the format specifications before writing it to `command`. If an attacker deliberately inserts dangerous formatted specifiers such as `%x` and `%n` into the string, it can lead to severe consequences, e.g., memory leakage, memory corruption, or even RCE. To patch this problem, we simply delete line 7. Again, DeepVulSeeker no longer reports the program as vulnerable after the patch, indicating that it is able to learn the insights of the formatted string problem.

```
1  static int ftp_file_size ( FTPContext * s )
2  {
3      char command [ CONTROL_BUFFER_SIZE ] ;
4      char * res = NULL ;
5      const int size_codes [ ] = { 213 , 0 } ;
6      snprintf ( command , sizeof ( command ) , " \r \n " , s -> path ) ;//delete
7      //body code
8  }
```

Fig. 8: Case 3

VI. RELATED WORKS

The research on software vulnerability indentification be divided into three categories, i.e., traditional detection methods, machine learning-based methods, and deep learning-based methods.

Traditional Detection Methods. Traditional detection methods refer to those that solely rely on fixed algorithms

or manual efforts to discover security deficiencies, without assistance from artificial intelligence. These methods are mainly static analysis, taint-tracking, symbolic execution, fuzzing, and dynamic analysis. Wu *et al.* employ static program analysis and natural language processing to differentially identify whether the vulnerability manifests a higher or lower severity in the target version [30]. Zhang *et al.* utilize taint-tracking to identify high-order vulnerabilities in OS kernels [31]. Kang *et al.* utilize the taint analysis to extract vulnerable traces and establish a signature database for detecting recurring vulnerabilities [32]. Yagemann *et al.* use symbolic execution to reconstruct program states and identify a couple of 0-day bugs [33]. Dinh *et al.* fuzzing the binding code of javascript to detect vulnerability [34]. Vadayath *et al.* exploit the advantages of both static and dynamic analysis to discover vulnerabilities in binary programs [30]. Even though These methods can achieve acceptable results, they either consume great labor costs or cannot achieve full automation.

Machine Learning-based Methods. Machine learning-based methods refer to those that only employ basic machine learning methods, instead of deep learning approaches, to identify bugs. Shi *et al.* use a random forest algorithm and SVM support vector machine algorithm [35]. Al-Yaseen *et al.* propose a multi-level hybrid intrusion detection model that uses a support vector machine and extreme learning machine to improve the efficiency of detecting known and unknown attacks [36]. Ren *et al.* propose DVCMA [37], a method for detecting software vulnerabilities based on clustering and model analyzing, and clustering technology is introduced to mine patterns from the set of vulnerability sequences in this method. Doffou *et al.* propose a model [38], which is developed by combining the Multiple Correspondence Analysis (MCA), the Elbow procedure, and the K-Means Algorithm. The problem with these methods is that they may result in poor performance when dealing with complicated codes.

Deep Learning-based Methods. Deep learning-based methods refer to those that leverage deep learning for identifying vulnerabilities. Zhou *et al.* [11] integrate multiple source code graph structural information to obtain code semantic representation. However, they do not use the pre-training mechanism to improve accuracy and some graph node information will be lost when the graph information is integrated. Tsankov *et al.* encode the dependency graph of Ethereum contracts to analyze smart contracts [39]. Nguyen *et al.* represent Ethereum smart contracts as heterogeneous contract graphs to detect new vulnerabilities accurately at the line-level and contract-level [40]. Liu *et al.* explore combining deep learning with expert patterns for smart contract vulnerability detection [13]. Tanwar *et al.* combine the effectiveness of recurrent, convolutional, and self-attention networks towards decoding the vulnerability hotspots in code [41]. Zhuang *et al.* explore using graph neural networks (GNNs) for smart contract vulnerability detection [42]. Dong *et al.* introduce deep-learning-based named entity recognition (NER) and relation extraction (RE) to recognize previous unseen software names and versions based on sentence structure

and contexts [43]. Feng *et al.* propose CodeBERT which is a bimodal model for programming language and natural language trained by Masked Language Modeling and Replaced Token Detection [10]. Guo *et al.* present GraphCodeBERT [44], which considers the inherent structure of code by Edge Prediction and Node Alignment to support tasks like code clone detection [45]–[47]. A variant of Roberta [48] in [21] supports both code understanding and generation. Alqarni *et al.* balance and fine-tune BERT [49] to low-level source code vulnerability detection [50]. Compared to current deep learning-based methods, our method not only takes advantage of the structural and semantic information of a program, but also provides more effective features due to the introduction of pre-trained models.

VII. Conclusions and Future Research

In this paper, we propose DeepVulSeeker, a novel model for code vulnerability identification. DeepVulSeeker is capable of acknowledging both structural information and semantic information from the codes by integrating the state-of-the-art technologies of pre-trained models, graph neural networks, and the self-attention mechanism. We test DeepVulSeeker on large heterogeneous datasets which contain conventional and highly-complicated codes, and the experimental results are more satisfactory than most of the cutting-edge baseline methods. We also conduct an ablation study and case study to further explore the model in depth. Even though our model outperforms other methods on QEMU and FFmpeg datasets, the accuracies and F1 still have rooms to be improved. We probe into the problem and find that the codes in these two datasets frequently contain inline assembly language, which cannot be handled by our method so far. In the future, we intend to revise our model with new algorithms to resolve the problem.

References

[1] Y. Xiao, Y. Jia, X. Cheng, J. Yu, Z. Liang, Z. Tian, I can see your brain: Investigating home-use electroencephalography system security, IEEE Internet of Things Journal 6 (4) (2019) 6681–6691. `doi:10.1109/JIOT.2019.2910115`.

[2] Z. Xu, T. Kremenek, J. Zhang, A memory model for static analysis of c programs, in: Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I, ISoLA'10, Springer-Verlag, Berlin, Heidelberg, 2010, p. 535–548.

[3] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, A. Tiu, Steelix: Program-state based binary fuzzing, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Association for Computing Machinery, New York, NY, USA, 2017, p. 627–637. `doi:10.1145/3106237.3106295`.
URL https://doi.org/10.1145/3106237.3106295

[4] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 529–540. `doi:10.1145/1315245.1315311`.
URL https://doi.org/10.1145/1315245.1315311

[5] J. Ming, D. Wu, G. Xiao, J. Wang, P. Liu, TaintPipe: Pipelined symbolic taint analysis, in: 24th USENIX Security Symposium (USENIX Security 15), USENIX Association, Washington, D.C., 2015, pp. 65–80.
URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ming

[6] D. Hin, A. Kan, H. Chen, M. A. Babar, LineVD: Statement-level Vulnerability Detection using Graph Neural Networks, arXiv e-prints (2022) arXiv:2203.05181 arXiv:2203.05181.

[7] T. Trong Vu, H. Dinh Vo, Using Multiple Code Representations to Prioritize Static Analysis Warnings, arXiv e-prints (2022) arXiv:2209.12181 arXiv:2209.12181.

[8] V. Nguyen, T. Le, C. Tantithamthavorn, J. Grundy, H. Nguyen, S. Camtepe, P. Quirk, D. Phung, An Information-Theoretic and Contrastive Learning-based Approach for Identifying Code Statements Causing Software Vulnerability, arXiv e-prints (2022) arXiv:2209.10414 arXiv:2209.10414.

[9] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, Z. Wang, Combining graph-based learning with automated data collection for code vulnerability detection, IEEE Transactions on Information Forensics and Security 16 (2021) 1943–1958. doi:10.1109/TIFS.2020.3044773.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

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

[12] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, J. Yin, Unixcoder: Unified cross-modal pre-training for code representation, in: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 7212–7225.

[13] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, S. Ji, Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion, in: Z.-H. Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21, International Joint Conferences on Artificial Intelligence Organization, 2021, pp. 2751–2759, main Track. doi:10.24963/ijcai.2021/379.
URL https://doi.org/10.24963/ijcai.2021/379

[14] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, P. S. Yu, Heterogeneous graph attention network, in: The world wide web conference, 2019, pp. 2022–2032.

[15] Y. Sun, J. Han, X. Yan, P. S. Yu, T. Wu, Pathsim: Meta path-based top-k similarity search in heterogeneous information networks, Proceedings of the VLDB Endowment 4 (11) (2011) 992–1003.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Advances in neural information processing systems 30 (2017).

[17] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush, Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Association for Computational Linguistics, Online, 2020, pp. 38–45.
URL https://www.aclweb.org/anthology/2020.emnlp-demos.6

[18] S. Bird, E. Klein, E. Loper, Natural language processing with Python: analyzing text with the natural language toolkit, " O'Reilly Media, Inc.", 2009.

[19] F. Chollet, et al., Keras, https://github.com/fchollet/keras (2015).

[20] M. Abadi, et al., Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 265–283.

[21] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[22] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, Z. Wang, Combining graph-based learning with automated data collection for code vulnerability detection, IEEE Transactions on Information Forensics and Security 16 (2020) 1943–1958.

[23] Nist software assurance reference dataset, https://samate.nist.gov/SARD/, (Accessed on 09/04/2022).

[24] S. Tomar, Converting video formats with ffmpeg, Linux Journal 2006 (146) (2006) 10.

[25] F. Bellard, Qemu, a fast and portable dynamic translator, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, USENIX Association, USA, 2005, p. 41.

[26] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, in: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, The Internet Society, 2018.
URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[27] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, 2014, pp. 1746–1751. doi:10.3115/v1/D14-1181.
URL https://aclanthology.org/D14-1181

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, CoRR abs/1706.03762 (2017). arXiv:1706.03762.
URL http://arxiv.org/abs/1706.03762

[29] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, Codebert: A pre-trained model for programming and natural languages, CoRR abs/2002.08155 (2020). arXiv:2002.08155.
URL https://arxiv.org/abs/2002.08155

[30] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang, et al., Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs, in: 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 413–430.

[31] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, Z. Qian, Statically discovering high-order taint style vulnerabilities in os kernels, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 811–824.

[32] W. Kang, B. Son, K. Heo, TRACER: signature-based static analysis for detecting recurring vulnerabilities, in: H. Yin, A. Stavrou, C. Cremers, E. Shi (Eds.), Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022, ACM, 2022, pp. 1695–1708. doi:10.1145/3548606.3560664.
URL https://doi.org/10.1145/3548606.3560664

[33] C. Yagemann, S. P. Chung, B. Saltaformaggio, W. Lee, Automated bug hunting with data-driven symbolic root cause analysis, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 320–336.

[34] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé, et al., Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases., in: NDSS, 2021.

[35] X. He, S. Liu, J. Jiang, Comparative study of intrusion detection methods based on machine learning, Netinfo Security 18 (5) (2018) 1–11.

[36] W. L. Al-Yaseen, Z. A. Othman, M. Z. A. 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 (2017) 296–303.

[37] J. Ren, B. Cai, H. He, C. Hu, A method for detecting software vulnerabilities based on clustering and model analyzing, Journal of Computational Information Systems 7 (04 2011).

[38] jeromeDiako Doffou, N. Gérard, M. O. Yapo, Modeling of a software vulnerability identification method, International Journal of Computer Science and Network Security 21 (9) (2021) 354–357.

[39] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018, pp. 67–82.

[40] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, L. Jiang, Mando: Multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities, arXiv preprint arXiv:2208.13252 (2022).

[41] A. Tanwar, H. Manikandan, K. Sundaresan, P. Ganesan, S. K. Chandrasekaran, S. Ravi, Multi-context attention fusion neural network for software vulnerability identification, CoRR abs/2104.09225 (2021).

[42] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, Q. He, Smart contract vulnerability detection using graph neural network., in: IJCAI, 2020, pp. 3283–3290.

[43] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, G. Wang, Towards the detection of inconsistencies in public security vulnerability reports, in: 28th USENIX security symposium (USENIX Security 19), 2019, pp. 869–885.

[44] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, in: International Conference on Learning Representations, 2020.

[45] C. Huang, H. Zhou, C. Ye, B. Li, Code clone detection based on event embedding and event dependency, in: Proceedings of the 13th Asia-Pacific Symposium on Internetware, Internetware '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 65–74. `doi: 10.1145/3545258.3545277`.
URL https://doi.org/10.1145/3545258.3545277

[46] X. Ji, L. Liu, J. Zhu, Code clone detection with hierarchical attentive graph embedding, International Journal of Software Engineering and Knowledge Engineering 31 (06) (2021) 837–861.

[47] A. Sheneamer, S. Roy, J. Kalita, An effective semantic code clone detection framework using pairwise feature fusion, IEEE Access 9 (2021) 84828–84844.

[48] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, V. Stoyanov, Roberta: A robustly optimized bert pretraining approach. arxiv 2019, arXiv preprint arXiv:1907.11692 (1907).

[49] J. D. M.-W. C. Kenton, L. K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of NAACL-HLT, 2019, pp. 4171–4186.

[50] M. Alqarni, A. Azim, Low Level Source Code Vulnerability Detection Using Advanced BERT Language Model, Proceedings of the Canadian Conference on Artificial Intelligence-Https://caiac.pubpub.org/pub/gdhb8oq4 (may 27 2022).