electronics

Article

Learning and Fusing Multi-View Code Representations for Function Vulnerability Detection

Zhenzhou Tian [1,2,3,*], Binhui Tian [1], Jiajun Lv [1] and Lingwei Chen [4]

[1] Department of Computer Science and Technology, Xi’an University of Posts and Telecommunications, Xi’an 710121, China; mx_info@126.com (B.T.); jjlv@stu.xupt.edu.cn (J.L.)
[2] Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi’an University of Posts and Telecommunications, Xi’an 710121, China
[3] Xi’an Key Laboratory of Big Data and Intelligent Computing, Xi’an University of Posts and Telecommunications, Xi’an 710121, China
[4] Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435, USA; lingwei.chen@wright.edu
* Correspondence: tianzhenzhou@xupt.edu.cn

Abstract: The explosive growth of vulnerabilities poses a significant threat to the security of software systems. While various deep-learning-based vulnerability detection methods have emerged, they primarily rely on semantic features extracted from a single code representation structure, which limits their ability to detect vulnerabilities hidden deep within the code. To address this limitation, we propose S^{2}FVD, short for Sequence and Structure Fusion-based Vulnerability Detector, which fuses vulnerability-indicative features learned from the multiple views of the code for more accurate vulnerability detection. Specifically, S^{2}FVD employs either well-matched or carefully extended neural network models to extract vulnerability-indicative semantic features from the token sequence, attributed control flow graph (ACFG) and abstract syntax tree (AST) representations of a function, respectively. These features capture different perspectives of the code, which are then fused to enable S^{2}FVD to accurately detect vulnerabilities that are well-hidden within a function. The experiments conducted on two large vulnerability datasets demonstrated the superior performance of S^{2}FVD against state-of-the-art approaches, with its accuracy and F1 scores reaching 98.07% and 98.14% respectively in detecting the presence of vulnerabilities, and 97.93% and 97.94%, respectively, in pinpointing specific vulnerability types. Furthermore, with regard to the real-world dataset D2A, S^{2}FVD achieved average performance gains of 6.86% and 14.84% in terms of accuracy and F1 metrics, respectively, over the state-of-the-art baselines. This ablation study also confirms the superiority of fusing the semantics implied in multiple distinct code views to further enhance vulnerability detection performance.

Keywords: vulnerability detection; representation fusion; attributed control flow graph

check for updates

Citation: Tian, Z.; Tian, B.; Lv, J.; Chen, L. Learning and Fusing Multi-View Code Representations for Function Vulnerability Detection. *Electronics* **2023**, *12*, 2495. https://doi.org/10.3390/electronics12112495

Academic Editors: Sanghyuk Lee, Soo Kyun Kim, Scott Uk-Jin Lee, Seokhun Kim and Asad Abbas

Received: 28 April 2023
Revised: 21 May 2023
Accepted: 30 May 2023
Published: 1 June 2023

1. Introduction

In recent times, the incidence of network attacks has witnessed a significant upsurge. These attacks are primarily driven by the ubiquitous presence of software vulnerabilities. To date, over 200,000 such vulnerabilities have been recorded on the Common Vulnerabilities and Exposures (CVE) website [1]. Given the pervasive exploitation of vulnerabilities and the significant security threats they pose, it is critical for developers to proactively detect vulnerabilities in their code. whether written by themselves or reused from open-source software.

However, identifying multi-faceted vulnerabilities requires security-related domain knowledge that goes beyond the expertise of most developers. This presents significant challenges for vulnerability detection. In light of the ever-expanding scale and complexity of modern software systems, it has become increasingly impractical, even for security professionals, to manually detect potential vulnerabilities within millions of lines of code, given the tremendous efforts and time required.

Inspired by its impressive performance in diverse domains such as NLP [2] and program analysis [3–6], deep learning has also been harnessed to develop a range of approaches [7–9] for the detection of vulnerabilities. These approaches utilize labeled training samples and extract semantic-aware features from them to construct classifiers that map the target code snippets onto a class space that indicates the absence or presence of vulnerabilities, or specific vulnerability types. Typically, prevailing deep-learning-based vulnerability detection methods rely on a single code representation structure to identify vulnerabilities, which, however, may fail to comprehensively capture vulnerability-indicative patterns and detect those vulnerabilities that are well-hidden within the code. This is due to the fact that these vulnerability-indicative patterns may require different perspectives on the code reflected by different code representation structures.

To address the aforementioned limitation, we propose S^2FVD, which entails a novel approach that leverages fused semantic vectors that are learned from three essential code representations, including token sequence, attribute control flow graph (ACFG), and abstract syntax tree (AST). These code representations provide distinct perspectives on the code, thereby allowing the model to more comprehensively capture vulnerability-indicative features from the code. The main contributions of this paper are summarized as follows.

- A novel DL-based vulnerability detection method called S^2FVD is presented. To accommodate the distinct representations of the code, an adaptive learning model has been devised to capture the multi-faceted aspects of function semantics and fuse them together to ensure the extraction of comprehensive semantic features. This strategy effectively prevents the loss of critical features that are indicative of vulnerability patterns.
- An extended-tree-structured neural network called ERvNN has been designed, which can effectively encode the semantics implied in the abstract syntax tree. With a GRU-style aggregation optimization on the tree nodes, it supports the straightforward and efficient encoding of multi-way tree structures, which otherwise should be firstly converted to the binary tree form.
- Extensive experiments were conducted to evaluate the performance of S^2FVD. The results demonstrated that S^2FVD outperformed existing state-of-the-art DL-based methods in terms of accuracy, F_1 score, precision, and recall when detecting the presence of vulnerabilities and pinpointing the specific vulnerability types. Moreover, ablation studies confirmed the effectiveness of the devised ERvNN for encoding AST and the strategy of representation fusion for enhancing the performance of S^2FVD.
- A new dataset has been constructed to facilitate vulnerability detection research. The dataset consists of 25,333 C functions, each of which is well labeled with either a specific CWE ID indicating a vulnerability or a non-vulnerable ground truth. The source implementation of the S^2FVD has also been made publicly available at https://github.com/lv-jiajun/S2FVD (accessed on 22 May 2023) to facilitate future benchmarking and comparisons.

The rest of this paper is structured as follows. Section 2 presents a review of closely related works. Section 3 delves into the essential designs of the S^2FVD by discussing the specific encoding of each distinct raw code view and the fusion strategies. The experimental evaluation details regrading the experimental setup, the evaluation results, and the observations of the S^2FVD and the comparison methods are outlined in Section 4. Section 5 discusses possible threats to validity issues, the limitations, and some interesting future works to extend. Finally, Section 6 concludes this work.

2. Related Work

The closely related vulnerability detection methods, which broadly fall into the following three categories, including code similarity-based methods [10,11], static-rule-based methods [12], and learning-based methods [13,14], are mainly discussed. Also, in introducing these methods, we focus more on the deep-learning-based ones. It should be noted that this is not a survey paper. Thus, the other types of vulnerability detection methods, which focus on binary code [15,16], examine executing dynamic analysis [17], or review formal semantic analysis [18,19] (e.g., model checking and symbolic execution), are not delved into.

2.1. Code-Similarity-Based Methods

Code-similarity-based vulnerability detection relies on the core idea that source code exhibiting high similarity is likely to share vulnerabilities [20,21]. However, while this approach can effectively identify vulnerabilities introduced through code cloning, it suffers from high rates of false negatives when it is used to detect other types of vulnerabilities not resulting from code cloning [22,23].

2.2. Rule-Based Methods

The static-rule-based methods involve scanning the target source code using a multitude of meticulously defined vulnerability rules or patterns. Prominent examples of typical static analyzers in this category include Infer [24], CodeChecker [25], and Checkmarx [26]. One of the main issues is that the vulnerability rules defined by human experts are often subjective, making it challenging to consider all possible scenarios that distinguish between vulnerabilities and non-vulnerabilities [27,28]. As a result, this approach may lead to a high rate of false positives and false negatives.

2.3. Learning-Based Methods

These methods can be broadly categorized as traditional machine- or deep-learning-based, depending on whether expert-defined features are required.

2.3.1. Conventional Machine Learning-Based Methods

Early works [29,30] typically utilized traditional machine learning algorithms for training detection models. These models rely on representative features that are engineered by experts such as code complexity metrics, code churns, imports and calls, and developer activities [31]. Nevertheless, these engineered features are often inadequate in indicating the presence of vulnerabilities. Additionally, most existing methods are restricted to in-project vulnerability detection, rather than providing general-purpose solutions.

2.3.2. Deep-Learning-Based Methods

Deep-learning-based methods, on the other hand, leverage the powerful feature learning capabilities of deep neural networks to automatically extract vulnerability patterns or features without requiring manual definition from experts [32]. The majority of deep-learning-based detection research concentrates on sequence-based code representation learning. For example, Russell et al. [7] developed a lexical analyzer to transform C/C++ functions into corresponding token sequences. These sequences were subsequently input into CNN and RNN models for training and then applied to detect code vulnerability. Li et al. created VulDeePecker [33], which is a vulnerability detection system based on deep learning. This system generates code gadgets (i.e., sets of control or data-dependent statements) that are lexically analyzed to establish token sequences, which are then fed into neural networks for vulnerability detection purposes. Later, Li et al. proposed SySeVR [8], which is a system framework for detecting vulnerabilities in C/C++ source code. This framework is primarily focused on obtaining code sequences that capture both syntactic and semantic information to achieve vulnerability detection.

Since sequence-based code representation overlooks the syntactic structure and control flow information inherent in source code, some research on code vulnerability detection has resorted to trees or graphs as code representations, as well as employing corresponding neural network models to learn semantic information within the code. For instance, Dam et al. [34] parsed a source code file into an abstract syntax tree and employed the

Tree-LSTM model to detect vulnerabilities within files. Zhou et al. [9] introduced Devign, which is a graph neural network (GNN) model that bases its composite code representation on the abstract syntax tree. Devign encodes various data and control dependencies to create a joint graph, which is subsequently input into GNNs to detect source code vulnerability. Li et al. [35] deployed a program dependence graph as the code representation and used a FA-GCN (graph convolution network with feature attention) to classify the graph, thereby achieving the successful detection of code vulnerabilities.

However, the single-representation-based method has difficulty in capturing the complete semantic information in the code, thus leading to higher rates of both false positives and false negatives. To address this issue, multiple distinct code representations are extracted, while adaptive deep neural network models are selected or devised to encode the different aspects of the function semantics. By retrieving the deeply implied semantic features and fusing them organically, more comprehensive vulnerability indicative features are obtained that lead to enhanced code vulnerability detection performance.

3. The Approach

The structural overview of the proposed approach is illustrated in Figure 1, wherein a function serves as the fundamental analysis unit, as opposed to an entire program, to assure a moderate detection granularity. S^{2}FVD first extracts and normalizes the token sequence, the ACFG, and the AST of the function as three raw code views of the function by parsing its lexical attributes and syntax. Next, word embedding is performed to derive initial vector representations for the tokens in the token sequence, the nodes in the ACFG, and the nodes in the AST. Subsequently, the vulnerability-indicative semantic features implied in each code view are captured using carefully selected or improved neural networks. Specifically, the token sequence is encoded using DPCNN, the ACFG is encoded using GAT, and the AST is encoded using an improved RvNN that is extended to support multi-way tree structures. Furthermore, these semantic features learned by these models are integrated in various ways to facilitate fusion. Finally, the fused vectors are classified in the classification layer to identify the presence of vulnerabilities or the specific vulnerability type.

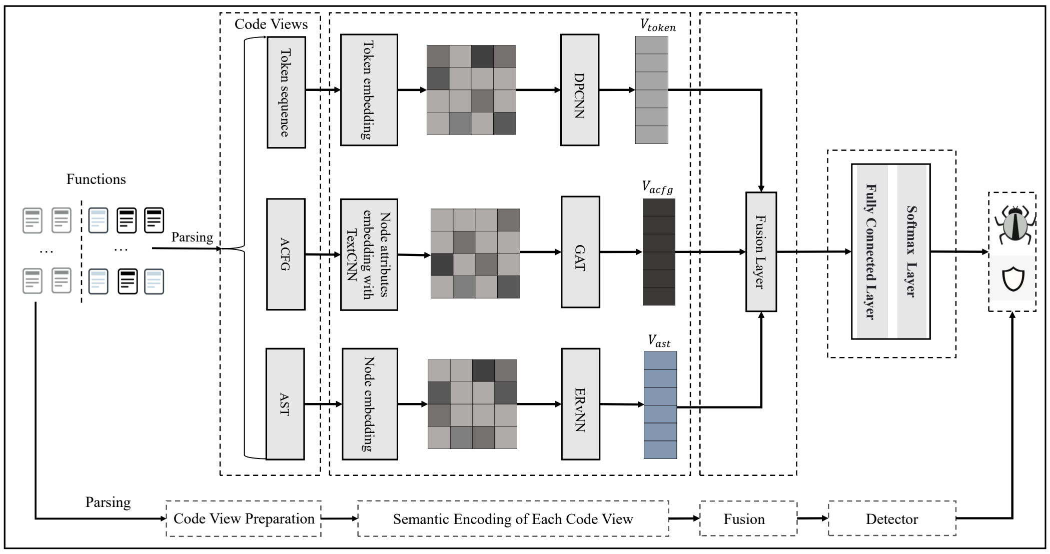


Figure 1. The overall architecture of S^{2}FVD for vulnerability detection.

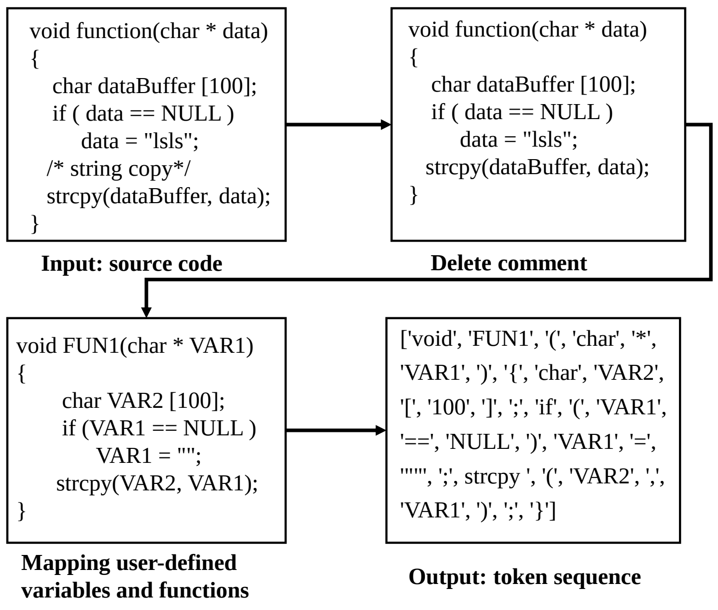
3.1. Semantic Encoding of Token Sequence

This section details the process of extracting semantic features from the token sequence. It covers the extraction and normalization of the token sequence from a function, as well as the specific neural network model employed to capture its semantics.

3.1.1. Token Sequence Preparation

The token sequence is processed in a manner similar to that performed in natural language processing. Such processing takes into account the natural order of the source code and reflects the programming logic embodied within the code to a significant degree.

Figure 2 presents the process of transforming a function into a token sequence, which entails the following operations. (1) Comment removal: Comments, being unrelated to code vulnerability, are removed from the function. (2) Code normalization: this involves screening self-defined variable and function names and replacing them with uniform names to remove semantically irrelevant information. Variables and function names within a function are mapped to corresponding symbol names in the order of their occurrence. For example, “VAR1” and “VAR2” represent different variables within the same function, and “FUN1” and “FUN2” represent different function names within the same program. Literals in strings are also removed, leaving only quotation marks. (3) Finally, lexing is performed to convert the pre-processed function into an ordered sequence of tokens, such as identifiers, keywords, operators, and symbols.



```
void function(char * data)
{
   char dataBuffer [100];
   if ( data == NULL )
       data = "lsls";
  /* string copy*/
  strcpy(dataBuffer, data);
}
```

```
void function(char * data)
{
   char dataBuffer [100];
   if ( data == NULL )
       data = "lsls";
  strcpy(dataBuffer, data);
}
```

```
void FUN1(char * VAR1)
{
      char VAR2 [100];
      if (VAR1 == NULL )
          VAR1 = "";
     strcpy(VAR2, VAR1);
}
```

```
['void', 'FUN1', '(', 'char', '*',
'VAR1', ')', '{', 'char', 'VAR2',
'[', '100', ']', ';', 'if', '(', 'VAR1',
'==', 'NULL', ')', 'VAR1', '=',
'""', ';', strcpy ', '(', 'VAR2', ',',
'VAR1', ')', ';', '}']
```

Figure 2. The process of transforming a function into a token sequence.

3.1.2. Sequence Encoding Network

The DPCNN [36] is well-known for its capability of capturing the long-range associations in sequences by augmenting the network depth. Given the fact that the token sequence of a function can be lengthy, the DPCNN model was adopted for extracting vulnerability-indicative features from the token sequence.

To detect whether a function is vulnerable using a learning model, it is necessary to convert the token sequence into a numerical vector. This facilitates the processing of the input for subsequent classifiers. In this regard, we employed the word2vec algorithm [37], which is a popular choice for producing high-quality token embeddings to convert these tokens into vectors, which are then fed into subsequent models for learning [38].

Figure 3 depicts the feature extraction of the token sequence through the DPCNN. On the basis of the token embedding obtained, each token sequence can be initially converted into a feature matrix A.

$$A = [e_1, e_2, \ldots, e_i, \ldots, e_l]^T \in R^{l \times d}, \tag{1}$$

where $e_i \epsilon R^d$ is the corresponding embedding of the ith token in the input sequence, l is the length of the token sequence, and d is the dimension of the token embedding. Specifically, in the implementation, the dimensions of the token embedding and length are set to 100 and 500, respectively.

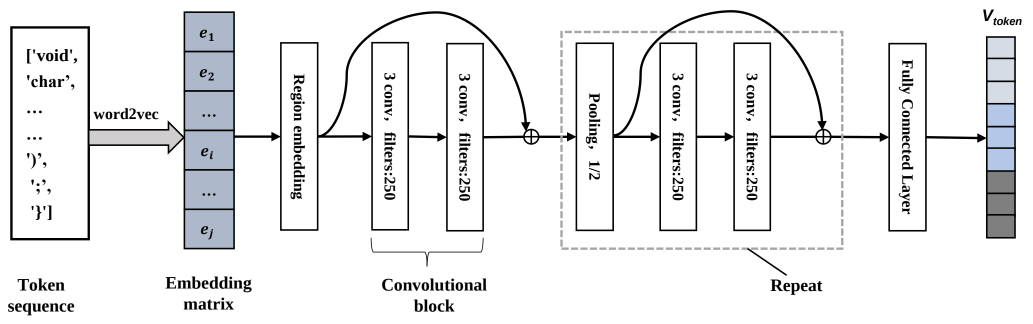


Figure 3. DPCNN-based semantics feature extraction from token sequences.

Subsequently, the region embedding operation is executed, where the token embedding matrix undergoes convolution processing using m filters with the dimensional size of $n \times d$. This generates a regional embedding capable of spanning multiple tokens. In the network implementation, n was set at 3 and m was set at 250. Two convolutional layers are then designated as convolutional blocks to conduct equal-length convolution operations, with the number of convolution filters and the size of the convolution kernels being fixed at 250 and 3, respectively. Following each convolutional block, a max pooling operation is performed with a stride size of two to compress the internal representation size of each function by half, thereby reducing the computational time for the subsequent convolution computations.

In addition, when initializing the DPCNN model, the initial weight values of each layer are typically small, which can impede the propagation of gradients. To address this issue, shortcut connections [39] were utilized, where the output obtained after the region embedding was added directly to the output obtained after the two equal-length convolution operations. The aggregated result is then passed as input to the subsequent layer of the network. Such connections help mitigate the impacts of small initial weights on each layer and prevent gradient vanishing. Formally, the shortcut optimization is defined as:

$$y = x + f(x), \tag{2}$$

where y denotes the output derived from two equal-length convolutions, and $f(x)$ is the input for the subsequent network layer.

Finally, the equal-length convolution and pooling operation are performed repeatedly to yield a single vector, which is then fed into a fully connected layer to obtain the feature vector V_{token} of the corresponding token sequence after undergoing linear transformation.

3.2. Semantic Encoding of ACFG

This section outlines the process of extracting semantic features from the ACFG. It includes the ACFG extraction process, as well as the graph neural network model used to encode its semantics.

3.2.1. ACFG Preparation

The control flow graph (CFG) is a commonly used code representation structure in the field of program analysis that implies semantic information regarding control dependencies between code elements. Additionally, the program statements within the control flow nodes are abstracted to assign attribute information to the nodes, thus enabling the construction of an attributed control flow graph (ACFG). This alternative code view of functions is able to capture not only the dependencies of control flow nodes, but also the attribute information associated with program statements.

The ACFG is a directed graph that can be represented as $G = (V, E, A)$, where V and E denote sets of vertices and edges, respectively, and A represents the set of attributes associated with each vertex in the graph. In the context of code vulnerability detection, each vertex corresponds to a node in the control flow graph, and each edge represents the control flow of the code. The attributes assigned to each node in the ACFG consist of both the node type and the specific program statement it represents. For instance, the node shown in Figure 4 with the "<operator>.equals" type signifies a logical operation, while the string "data==null" corresponds to the detailed program statement within the node.

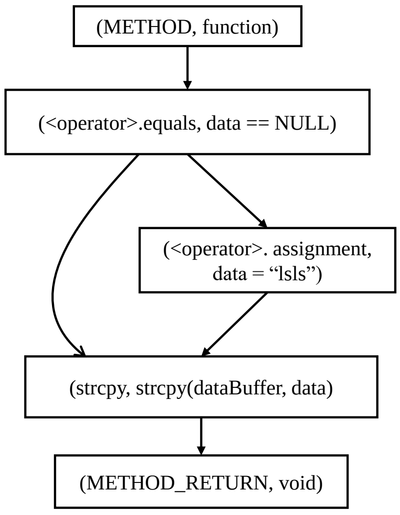

(METHOD, function)
(<operator>.equals, data == NULL)
(<operator>. assignment, data = "lsls")
(strcpy, strcpy(dataBuffer, data)
(METHOD_RETURN, void)

Figure 4. An illustrative example of the attributed control flow graph.

Similarly, to facilitate the handling of node attributes in the subsequent encoder network, a lexical analysis is conducted to convert node attributes into token sequences. For instance, the node attributes "strcpy, strcpy (dataBuffer, data)" can be represented as a sequence of eight tokens, i.e., "strcpy", ",", "strcpy", "(", "dataBuffer", ",", "data", and ")". Specifically, to extract the attributed control flow graph from the source code of C functions, the widely used open-source tool Joern [40] is utilized.

3.2.2. Graph Encoding Network

To deal with the ACFG, either a graph attention network (GAT) [41] or a graph convolutional network (GCN) [42] can be leveraged for its representation learning. The essence of both GAT and GCN is to generate more expressive representations for nodes by aggregating features from their own nodes alongside their neighboring nodes. Different from GCN, the GAT model assigns different weights to different nodes in the same neighborhood, which is believed to promote more effective integration of the inter-node feature correlations and ultimately enhances overall feature extraction performance. Therefore, GAT was selected as the base neural network structure for extracting semantic features from the ACFG.

In order to effectively extract the semantic information contained in the individual nodes, the TextCNN model, which is particularly well-suited for capturing the distinct features of tokens by applying different convolution kernels, is used to extract the initial features of the nodes in the ACFG. Specifically, the token sequence that corresponds to the attributes of each node is transformed into a feature matrix $N \in R^{w \times k}$, where w is the token embedding dimension, and k represents the length of the token sequence. Thereafter, feature extraction is carried out via convolution kernels of sizes two, three, and four, respectively. Since the feature maps obtained from convolution kernels of different sizes may have different dimensions, a pooling function is employed to standardize their dimensions. Finally, the resulting representations are concatenated and transformed into output features through a fully connected layer. These features serve as the initial features for the nodes in the graph, which are then processed by the GAT model.

Figure 5 illustrates the ACFG feature extraction structure built on the GAT. The vector VEC_i produced by TextCNN served as the initial hidden state of node B_i. These initial hidden states were organized into an $n \times m$-dimensional feature matrix X, while the connections between nodes constituted an $n \times n$-dimensional matrix A, known as an adjacency matrix. Here, n denotes the number of nodes, and m denotes the dimension of each node's initial hidden state. X and A were fed as inputs into the GAT model for attention-enhanced hidden feature aggregation, which can be formally expressed as follows:

$$X^{(l+1)} = f(X^{(l)}, A) \tag{3}$$

$$x_i^{(l+1)} = ||_{k=1}^{K} \sigma(\sum_{j \epsilon N_i} a_{ij}^k W^k x_j^{(l)}) \tag{4}$$

where X^l is the hidden state of the nodes at layer l, $x_i^{(l+1)}$ is the hidden state of node i at layer $l+1$, $x_j^{(l)}$ is the hidden state of all the neighboring nodes of node i at layer l, W^k denotes the corresponding linear transformation matrix for input features, and a_{ij}^k denotes the weights for the kth group of attention mechanisms.

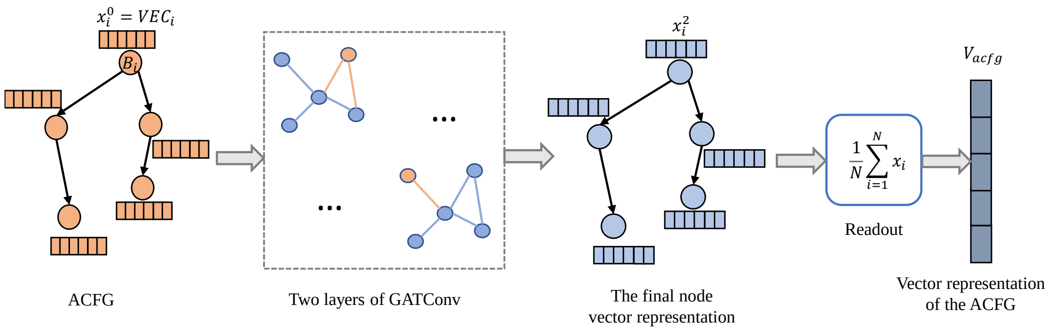

$x_i^0 = VEC_i$
B_i
ACFG
...
Two layers of GATConv
x_i^2
The final node vector representation
$\frac{1}{N}\sum_{i=1}^{N} x_i$
Readout
V_{acfg}
Vector representation of the ACFG

Figure 5. GAT-based semantic feature extraction from ACFG.

In the specific implementation, S^{2}FVD features a two-layer GAT structure. The first layer consists of multi-head attention, and the second layer consists of single-head attention. The utilization of multiple attention layers facilitates effective learning of the deep semantic features. As such, three independent groups of attention mechanisms are employed in the first layer, and their outputs are subsequently concatenated to obtain x_i^1. For the readout operation, the graph representation V_{acfg} is computed by taking the mean of the node representations as follows:

$$x_g = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{5}$$

where N denotes the number of nodes in the ACFG, and x_i is the feature vector of node i.

3.3. *Semantic Encoding of AST*

This section presents the details for extracting semantic features from the AST. This encompasses the AST preparation phase and the proposed extended tree-structured neural network, which endorses the direct encoding of multi-way tree structures through a GRU-style aggregation optimization for the tree nodes.

3.3.1. AST Preparation

The Abstract Syntax Tree [43] is another widely used code representation in program analysis, where the primary code elements (e.g., variable types, symbols, and operators) constitute its leaves, and the defined set of code structures (e.g., expressions and loops) constitutes its non-leaf nodes. The AST depicts both the lexical information and the syntactic structures of the source code [44].

Other code representations such as a program dependency graph (PDG) are artificially constructed and tend to emphasize specific facets of the code (e.g., dependencies between statements), which may suffer from semantic distortion or loss when representing incomplete or non-compilable code fragments. By contrast, the AST stands out as a lossless code representation that preserves the naturalness of the code, thereby yielding more comprehensive and precise semantics than these other representations. Furthermore, a previous investigation [45] has suggested that the AST is a superior code representation for detecting vulnerabilities. Specifically, to acquire the ASTs for functions, the well-established C parsing library Pycparser [46] is utilized. Figure 6 illustrates an instance of the parsed AST.

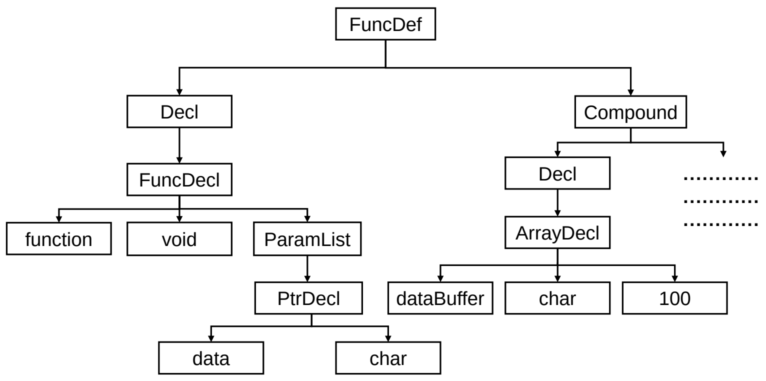


Figure 6. An illustrative example of AST.

3.3.2. Tree Encoding Network

Recursive neural networks (RvNNs) were adopted to extract features from tree-structured data. Their core idea is to recursively generate feature vectors for each node in the tree by aggregating the features of its child nodes. However, a standard tree-structured RvNN [47] only deals with binary trees and cannot directly handle the typical multi-way tree structure of ASTs. Thus, this work extended the aggregation operation to support multiple child nodes as inputs, which we referred to as the Extended Recursive Neural Network (ERvNN). The ERvNN served as the neural network model for encoding the semantics from ASTs.

Let us consider an abstract syntax tree $\mathcal{T} = \{V, E\}$, where V and E denote its node and edge sets, respectively. For a given node $v_i \in V$, let $\mathcal{S}_i$ denote its immediate child nodes. Then, the hidden state of the node v_i is computed through a GRU-style neural unit by integrating the semantic information of both the child nodes and the node v_i itself. This computation can be formulated as:

$$\mathbf{r}_i = \sigma(\mathbf{W}_r \mathbf{h}_{\mathcal{S}} + \mathbf{U}_r \mathbf{e}_i + \mathbf{b}_r), \tag{6}$$

$$\mathbf{z}_i = \sigma(\mathbf{W}_z \mathbf{h}_{\mathcal{S}} + \mathbf{U}_z \mathbf{e}_i + \mathbf{b}_z), \tag{7}$$

$$\widetilde{\mathbf{h}}_i = tanh(\mathbf{W}_h(\mathbf{r}_i \odot \mathbf{h}_{\mathcal{S}}) + \mathbf{U}_h \mathbf{e}_i + \mathbf{b}_h), \tag{8}$$

$$\mathbf{h}_i = \mathbf{z}_i \odot \mathbf{h}_{\mathcal{S}} + (1 - \mathbf{z}_i) \odot \widetilde{\mathbf{h}}_i, \tag{9}$$

where $\mathbf{h}_{\mathcal{S}}$ signifies the semantics aggregated from the children by max pooling the hidden states of all the nodes in $\mathcal{S}_i$; σ represents the sigmoid activation function; $\odot$ denotes the element-wise product operation, $\mathbf{W}$s, $\mathbf{U}$s, and $\mathbf{b}$s are the weights and biases that need to be learned during the model training process, respectively; $\mathbf{e}_i$ is the embedding vector that corresponds to the token in node v_i, which can be obtained via looking up the token embeddings that have been pre-trained with the word2vec algorithm.

After iteratively calculating the hidden state of each node in the AST using ERvNN in a bottom-up way, the hidden state of its root node is taken as the AST's final semantic vector representation. In this regard, given an AST, its semantic encoding can be denoted as $V_{ast} = \mathbf{h}_{v_0}$, where $\mathbf{h}_{v_0}$ is the hidden state of the AST's root node. Figure 7 illustrates the layer-by-layer semantic aggregation process for encoding the entire tree.

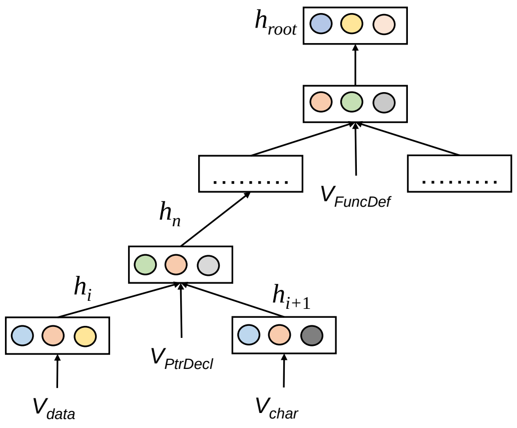


Figure 7. ERvNN-based semantic feature extraction from AST.

3.4. Multi-View Fusion

To capture more comprehensive semantic information from the program code—the semantic vector V_{acfg} that is obtained using the GAT on the attributed control flow graph—the semantic vector V_{token} that is obtained using the DPCNN on the function token sequence, and the semantic vector V_{ast} that is obtained using the proposed ERvNN on the abstract syntax tree, are further fused.

There are various strategies for fusing the vectors, and, in this work, five widely-used approaches were considered, including point-by-point addition, concatenation, average pooling, max pooling, and non-linear fusion with a multiple layer perceptron (MLP). These strategies were empirically evaluated to determine the best one for our task. Formally, the fused representations can be computed as:

$$V_{add} = \{v_i | v_i = V^i_{token} + V^i_{acfg} + V^i_{ast}, i = 1, 2, \ldots, n\} \tag{10}$$

$$V_{avg} = \{v_i | v_i = \mathrm{avg}(V^i_{token}, V^i_{acfg}, V^i_{ast}), i = 1, 2, \ldots, n\} \tag{11}$$

$$V_{max} = \{v_i | v_i = \max(V^i_{token}, V^i_{acfg}, V^i_{ast}), i = 1, 2, \ldots, n\} \tag{12}$$

$$V_{con} = [V_{token}; V_{acfg}; V_{ast}] \tag{13}$$

$$V_{mlp} = \mathrm{MLP}_\theta[V_{token}; V_{acfg}; V_{ast}] \tag{14}$$

where V_{add}, V_{avg}, V_{max}, V_{con}, and V_{mlp} denote the fused vector obtained with the point-by-point addition strategy, the average pooling, the max pooling, the concatenation strategy, and the MLP, respectively. It is worth emphasizing that the addition, average, and max pooling operations between the participant vectors require the same dimensionality. In the specific implementation, the dimensions of the extracted feature vectors V_{token}, V_{acfg}, and V_{ast} were all equal to 192, which otherwise should be padded to the same length accordingly.

4. Experiments and Evaluations

To evaluate the effectiveness of S^{2}FVD, the following research questions were explored:

- RQ1: Impacts of the Fusion Strategies—which fusion strategy, as discussed in Section 3.4, most effectively blends the semantic features collected from the distinct code perspectives for S^{2}FVD to deliver its best vulnerability detection performance?
- RQ2: Performance Comparison with Baseline Methods—how does the performance of S^{2}FVD compare to the baseline methods in detecting the presence of vulnerabilities, as well as pinpointing the specific vulnerability types?

- RQ3: Substitutional Study—how does S^{2}FVD behave when its constituent neural network structures are substituted with other typical neural networks?
- RQ4: Ablation Study—does fusing multiple semantic features captured from distinct code views help boost the vulnerability detection performance compared with using part of them?

4.1. Experimental Setup

The datesets used for the evaluation, the experiment settings regarding the model training and testing, the baseline methods against which S^{2}FVD were compared against, as well as the evaluation metrics, are described in this section.

4.1.1. Datasets

To evaluate the proposed method, we constructed a dataset consisting of C functions on the basis of the Software Assurance Reference Dataset (SARD) [48], which is a vulnerability database that is widely used as a source for producing experimental samples. The programs in SARD consist of a blend of academic, production, and synthetic code, with each program categorized as "bad", "good", or "mixed". Typically, each "bad" program contains one vulnerable function, while each "good" program comprises fixed or patched non-vulnerable functions. A "mixed" program contains both a vulnerable function and its patched versions within a single program.

The C source file is typically composed of a header file, macro definition statements, and multiple functions. To generate the function samples, the ANTLR tool [49] was used to parse the raw C source files. Initially, the source file is read, and macro expansion is performed during preprocessing to replace macro names with strings, as macros may contain vulnerability-related information. Subsequently, the source file is transformed into the ANTLR file stream format, which serves as input for the subsequent lexical analysis phase. Given that C programs were being dealt with, CPP14Lexer was used for lexing, thereby producing a sequence of matching tokens. The token sequence was then passed to the parser for syntactic analysis, which converts the program into a syntax tree to facilitate the extraction of the hierarchical structure of the program. During the syntax tree traversal, each node was examined, and an instance with the type of "FunctionDefinitionContext" was marked as the root node of a function subtree. By traversing the subtree in a depth-first manner, the specific source code regarding the function within the source file could be extracted.

Finally, a total of 13,541 non-vulnerable functions and 11,792 vulnerable functions were gathered, which are scattered in 26 distinct types of vulnerabilities. Table 1 presents in detail the number of functions that locates in each vulnerability type, along with their corresponding labels, from the set of 11,792 vulnerable functions.

Table 1. The number of vulnerable functions corresponding to different vulnerability types.

CWE ID	Number	Label	CWE ID	Number	Label
CWE78	1243	0	CWE197	231	13
CWE90	142	1	CWE252	205	14
CWE114	166	2	CWE253	329	15
CWE121	1599	3	CWE369	233	16
CWE122	1135	4	CWE400	195	17
CWE124	485	5	CWE401	361	18
CWE126	407	6	CWE427	136	19
CWE127	485	7	CWE457	259	20
CWE134	530	8	CWE590	233	21
CWE190	1138	9	CWE606	160	22
CWE191	881	10	CWE690	320	23
CWE194	297	11	CWE761	190	24
CWE195	297	12	CWE789	135	25

Since the programs in SARD are basically synthetic, we also evaluated S^{2}FVD and the comparison works against a real-world vulnerability dataset called D2A [50]. This dataset was curated by the IBM research team from multiple popular open-source software projects, including FFmpeg, httpd, Libav, LibTIFF, Nginx, and OpenSSL.

4.1.2. Experiment Settings

To conduct the experiments, both datasets were partitioned into the training, validation, and testing sets using an 8:1:1 proportion. The models were then trained with an initial learning rate of 1×10^{-3}, which was reduced by 0.8 after every 10 epochs using the Adam optimizer and a batch size of 16. In each epoch, the training set was shuffled, and accuracy on the validation set was computed. The early stopping mechanism was used to halt the training when the validation accuracy did not improve after 5 epochs. The model that achieved the best accuracy was then selected as the final detection model, which was used to evaluate the performance on the testing set. All the experiments were conducted on two Linux servers, each equipped with two 2.1 GHz Intel Xeon Silver-4310 CPUs, 128 GB RAM, and two NVIDIA RTX3090 GPUs.

4.1.3. Baseline Methods

Three state-of-the-art deep-learning-based vulnerability detection methods, including VulDeePecker, SySeVR, and Reveal, were used as the comparison baselines. A brief overview of them is presented below:

- VulDeePecker proposes to extract code gadgets, which are comprised of code statements that exhibit control dependency relationships with respect to certain code elements of interest (such as library/API calls and array usage), to represent programs. Recurrent neural networks are then trained on these gadgets to detect vulnerabilities.
- SySeVR further enriches the concept of code gadgets. It proposes SeVCs (semantic-based vulnerability candidates) to represent the code by taking into account the data dependencies among the code statements in addition to the control dependencies.
- Reveal is an approach that operates on the graph-based representation of code known as the code property graph (CPG). It uses a GGNN (gated graph neural network) to extract features that are indicative of vulnerabilities present in the code.

4.1.4. Evaluation Metrics

As with most existing learning-based methods for vulnerability detection, the widely used metrics of accuracy, precision, recall, and F1-score were adopted to evaluate the performance of S^{2}FVD and the comparison methods. It should be noted that "Accuracy" denotes the overall accuracy, while "Precision", "Recall", and "F1-score" correspond to the weighted-averages of precision, recall, and F1-score, respectively, in the experiments involving the detection of vulnerability types.

To be specific, let k denote the class label, let $\{c_1, c_2, \cdots, c_k\}$ denote the number of function samples for each class, and let $\{c'_1, c'_2, \cdots, c'_k\}$ be the number of functions whose corresponding class was accurately classified by the classifier. The accuracy can be defined as follows:

$$\text{Accuracy} = \frac{\sum_{i=1}^{k} c'_i}{\sum_{i=1}^{k} c_i} \tag{15}$$

Let $\{p_1, p_2, \cdots, p_k\}$, $\{r_1, r_2, \cdots, r_k\}$, and $\{f_1, f_2, \cdots, f_k\}$ be the precision, recall, and F$_1$-score values computed with respect to the k classes, respectively. The weighted-average precision, recall, and F$_1$-score can be defined as:

$$\text{Precision} = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} p_i, \qquad \text{Recall} = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} r_i, \qquad \text{F}_1 = \sum_{i=1}^{k} \frac{c_i}{\sum_{j=1}^{k} c_j} f_i \tag{16}$$

4.2. Experimental Results

The subsequent sections cover the experimental findings pertaining to the research questions as discussed at the beginning of Section 4.

4.2.1. RQ1: Impacts of the Fusion Strategies

To identify the most effective fusion strategy that best enhanced the vulnerability detection capability of S^2FVD, its performances under different fusion strategies were evaluated in this experiment.

As summarized in Table 2, the values of the performance metrics, S^2FVD_{mlp}, which resorted to a MLP for feature fusion, outperformed the alternative models that adopted the other fusion strategies, in both the vulnerability presence detection and in the vulnerability type detection task. This can be attributed to several reasons: Firstly, the concatenation of the feature vectors preserved more semantic information than straightforwardly averaging or adding them in a point-by-point manner, as the extracted code representations provided complementary descriptions of the functions from both the sequence and structure perspectives; Additionally, the non-linear fusion capability provided by the MLP empowered S^2FVD to pay more attention to the vulnerability indicative features from the concatenated vectors, which, therefore, made it more advantageous for vulnerability detection.

Table 2. Impacts of the fusion strategies on the vulnerability detection capability of S^2FVD.

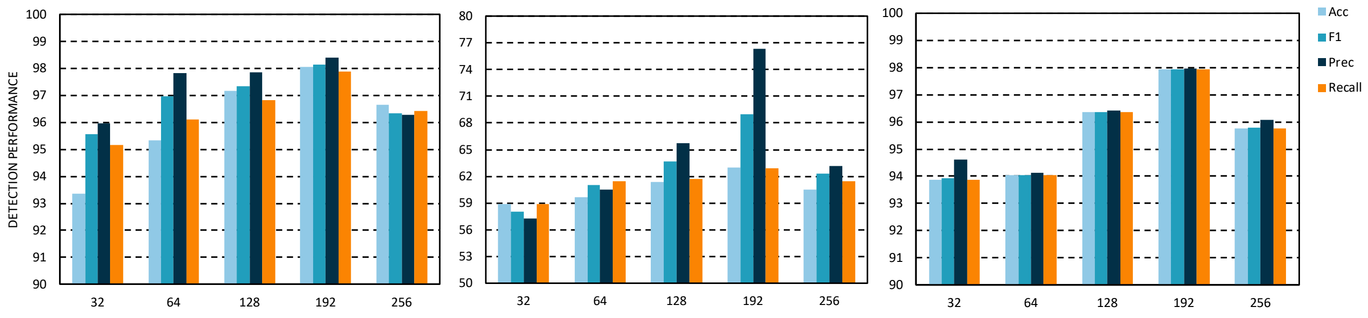
(a) Performance on Vulnerability Presence Detection

Method	Dataset	Accuracy	F_1	Precision	Recall
S^2FVD_{add}	SARD	96.04	96.11	96.34	95.88
S^2FVD_{con}		96.58	96.58	96.99	96.17
S^2FVD_{avg}		96.30	96.30	96.63	95.98
S^2FVD_{max}		97.28	97.28	97.63	96.93
S^2FVD_{mlp}		**98.07**	**98.14**	**98.41**	**97.88**
S^2FVD_{add}	D2A	58.97	58.12	56.46	59.88
S^2FVD_{con}		58.87	61.04	60.00	62.12
S^2FVD_{avg}		58.10	55.99	52.85	59.53
S^2FVD_{max}		60.69	66.17	71.61	61.50
S^2FVD_{mlp}		**63.02**	**68.99**	**76.30**	**62.95**

(b) Performance on Vulnerability Type Detection

Method	Dataset	Accuracy	Weighted F_1	Weighted Precision	Weighted Recall
S^2FVD_{add}	SARD	95.19	95.04	95.26	95.19
S^2FVD_{con}		96.49	96.49	96.54	96.49
S^2FVD_{avg}		96.02	96.02	96.19	96.02
S^2FVD_{max}		97.11	97.11	97.15	97.11
S^2FVD_{mlp}		**97.93**	**97.94**	**97.97**	**97.93**

As one may have noticed, the fusion process utilizing the MLP can alter the dimensionality of the semantic encoding vectors. Intuitively speaking, the output vector with an improperly small size could lead to the loss of subtle code semantics. On the other hand, retaining an approximate dimension as the input vector does not necessarily improve the fusion effect, but it does increase the computational costs. Therefore, in order to gain insight into the effects of this hyper-parameter on the fusion process, the detection performances of S^2FVD_{mlp} parameters were evaluated by varying the dimensionality of the fused semantic vectors. As illustrated in Figure 8, $S^2FVD\textit{mlp}$ exhibited optimal performance at a dimensionality of 192. Thus, for the sake of simplicity, in the following experiments, unless explicitly stated, S^2FVD will always refer to S^2FVD_{mlp}, with the fused vector's dimensionality set to 192.



(a) Presence Detection on SARD (b) Presence Detection on D2A (c) Type Detection on SARD

Figure 8. Dimensionality impacts of the fused vector on the detection performance.

4.2.2. RQ2: Performance Comparison with Baseline Methods

In this experiment, the efficacy of S^{2}FVD in either detecting the presence of vulnerabilities or in pinpointing the specific vulnerability types was assessed and compared with the SOTA baseline approaches, as discussed in Section 4.1.3. The evaluation results are presented in Table 3. As the metric values show, S^{2}FVD demonstrated a performance that was superiority regarding both the vulnerability detection tasks and in terms of all the metrics. This indicates S^{2}FVD's ability to effectively capture the significant vulnerability features encoded in the fused semantic vectors in a more comprehensive and precise manner.

Table 3. Performance comparison with the DL-based baseline methods.

(a) Performance regarding vulnerability presence detection					
Method	Dataset	Accuracy	F$_1$	Precision	Recall
VulDeePecker	SARD	95.06	95.01	95.41	94.62
SySeVR		96.57	96.67	96.88	96.46
Reveal		96.55	96.42	96.64	96.21
S^{2}FVD		**98.07**	**98.14**	**98.41**	**97.88**
VulDeePecker	D2A	56.47	60.86	66.11	56.39
SySeVR		60.50	60.76	59.74	61.82
Reveal		60.12	58.65	55.24	62.50
S^{2}FVD		**63.02**	**68.99**	**76.30**	**62.95**
(b) Performance on Vulnerability Type Detection					
Method	Dataset	Accuracy	Weighted F$_1$	Weighted Precision	Weighted Recall
VulDeePecker	SARD	95.38	95.34	95.60	95.38
SySeVR		96.20	96.08	96.34	96.20
Reveal		95.79	95.69	95.97	95.79
S^{2}FVD		**97.93**	**97.94**	**97.97**	**97.93**

Specifically, regarding vulnerability presence detection, S^{2}FVD achieved a leading accuracy of 98.07% and an F1-score of 98.14% for the SARD dataset, as well as an accuracy of 63.02% and and F1-score of 68.99% for the D2A dataset. For the vulnerability type detection task, where the methods are required to identify the specific vulnerability type present in the vulnerable code, S^{2}FVD again demonstrated the best performance among the comparison methods, with an accuracy of 97.93% and an F1-score of 97.94%. In addition, it can be observed that the performance results of the DL-based approaches on the D2A dataset were much lower than on the synthetic dataset, thereby suggesting that detecting vulnerabilities in real-world programs is still challenging, due to the more intricate and varied code contexts that vulnerabilities reside in. However, S^{2}FVD exhibited promising performance gains compared to other DL-based methods, with an average improvement in detection accuracy and F1-score of 6.86% and 14.84%, respectively. This emphasizes the potential of S^{2}FVD in detecting vulnerabilities, even in much more complex code contexts.

Table 3 shows that S^{2}FVD exhibited generally superior performance compared to the baseline methods, as evidenced by the metric values. To further verify whether this performance difference was statistically significant, the Wilcoxon rank sum test and *t*-test were enforced between S^{2}FVD and each of the baseline methods using 5×2 cross-validation. The *p*-values for accuracy and the comprehensive metric F1-score are presented in Table 4. It can be observed that none of the *p*-values exceeded 0.05 for either test, thereby indicating that there was a statistically significant difference between S^{2}FVD and the baseline methods.

Table 4. Statistical significance testing between S^{2}FVD and the baseline methods.

(a) Results for vulnerability presence detection					
Dataset	Method Pair	Wilcoxon Random-sum		*t*-test	
		Acc. *p*-value	F_1 *p*-value	Acc. *p*-value	F_1 *p*-value
SARD	S^{2}FVD vs. VulDeePecker	0.0090	0.0090	0.0003	0.0008
	S^{2}FVD vs. SySeVR	0.0163	0.0163	0.0123	0.0139
	S^{2}FVD vs Reveal	0.0090	0.0090	0.0002	0.0004
D2A	S^{2}FVD vs. VulDeePecker	0.0090	0.0162	0.0001	0.0308
	S^{2}FVD vs. SySeVR	0.0090	0.0472	0.0001	0.0280
	S^{2}FVD vs. Reveal	0.0090	0.0125	0.0004	0.0362
(b) Results on Vulnerability Type Detection					
SARD	S^{2}FVD vs. VulDeePecker	0.0090	0.0090	0.0002	0.0002
	S^{2}FVD vs. SySeVR	0.0080	0.0080	0.0004	0.0003
	S^{2}FVD vs. Reveal	0.0090	0.0090	0.0002	0.0002

4.2.3. RQ3: Substitutional Analysis

In this section, we conducted substitutional experiments by replacing the constituent neural network structures used in S^{2}FVD to extract semantic features from the different code views with other typical neural networks. Specifically, we selected three other sequence-oriented models, including TextCNN, TextRNN, and Transformer [51], to encode the token sequences, in addition to the originally adopted DPCNN in S^{2}FVD. These models are well-known for their superior feature capturing capability in handling sequences. For extracting features from the ACFG, a graph convolution neural network (GCN) was regarded as the substitute of the GAT for performance comparison. For extracting features from the AST, TBCNN [52] was selected as the substitute of the original ERvNN for comparison.

The results obtained on the vulnerability datasets, as shown in Table 5, indicate that the combination of the encoding models utilized in S^{2}FVD resulted in the best performing vulnerability detection model, and substituting different parts of it with the listed alternatives led to varying degrees of performance degradation. Additionally, the metric values of S^{2}FVD$_{TBCNN}$, where TBCNN was substituted for ERvNN to encode the AST, suggest that our designed ERvNN can capture the semantics implied in the AST more effectively.

4.2.4. RQ4: Ablation Study

To ascertain whether the fusion of multiple semantic vectors encoded from the distinct code views contributes to the enhanced performance of S^{2}FVD in detecting vulnerabilities compared to utilizing only a subset of them (i.e., utilizing a single vector or the vector fused from any two code views), an ablation study was conducted in this experiment.

In Table 6, the experimental results show that the overall performance of S^{2}FVD surpassed the alternative models that only fuse semantic vectors extracted from two types of code views. Moreover, these alternative models outperform the ones that solely utilize the semantic vector obtained from a single code view. The progressive improvement in performance as the number of distinct code views was increased indicates the effectiveness of the fusion strategy in combining the semantic features extracted from diverse aspects of the code. These findings also suggest that, when the model is trained with fewer representations, it may struggle to fully comprehend the semantic information implied in the code, thereby resulting in inferior detection outcomes. It can also be inferred that there may be some degree of semantic overlap between the features extracted from the token sequence, the attributed control flow graph, and the abstract syntax tree. However, by additionally identifying and fusing the disjointed parts of these features, the vulnerability detection capability of the model was substantively enhanced.

Table 5. Substitutional analysis of the constituent neural network structures in S^{2}FVD.

(a) Performance for vulnerability presence detection

Method	Dataset	Accuracy	F_1	Precision	Recall
S^{2}FVD$_{TextCNN}$	SARD	96.53	96.61	96.79	96.44
S^{2}FVD$_{TextRNN}$		96.96	96.69	97.03	96.35
S^{2}FVD$_{Transformer}$		97.18	97.04	97.26	96.82
S^{2}FVD$_{GCN}$		96.26	96.35	96.69	96.02
S^{2}FVD$_{TBCNN}$		97.58	97.78	97.94	97.62
S^{2}FVD		**98.07**	**98.14**	**98.41**	**97.88**
S^{2}FVD$_{TextCNN}$	D2A	60.57	63.73	67.56	60.31
S^{2}FVD$_{TextRNN}$		60.41	63.96	67.97	60.40
S^{2}FVD$_{Transformer}$		61.05	64.13	66.07	62.31
S^{2}FVD$_{GCN}$		61.00	62.92	64.09	61.79
S^{2}FVD$_{TBCNN}$		62.15	63.15	63.90	62.42
S^{2}FVD		**63.02**	**68.99**	**76.30**	**62.95**

(b) Performance for vulnerability type detection

Method	Dataset	Accuracy	Weighted F_1	Weighted Prec.	Weighted Rec.
S^{2}FVD$_{TextCNN}$	SARD	96.20	96.24	96.39	96.20
S^{2}FVD$_{TextRNN}$		95.94	95.99	96.16	95.94
S^{2}FVD$_{Transformer}$		96.32	96.35	96.53	96.32
S^{2}FVD$_{GCN}$		96.09	96.14	96.29	96.09
S^{2}FVD$_{TBCNN}$		96.24	96.25	96.48	96.24
S^{2}FVD		**97.93**	**97.94**	**97.97**	**97.93**

Table 6. Ablation study of the semantic vectors encoded from the distinct code views.

(a) Performance for vulnerability presence detection.

Method	Dataset	Accuracy	F_1	Precision	Recall
S^{2}FVD$_{DPCNN}$	SARD	87.95	89.15	92.31	86.20
S^{2}FVD$_{GAT}$		88.60	88.58	90.65	86.60
S^{2}FVD$_{ERvNN}$		89.06	90.35	97.01	84.55
S^{2}FVD$_{DPCNN+GAT}$		92.12	91.99	91.66	92.33
S^{2}FVD$_{DPCNN+ERvNN}$		92.67	92.47	92.21	92.74
S^{2}FVD$_{GAT+ERvNN}$		94.33	94.20	93.88	94.53
S^{2}FVD		**98.07**	**98.14**	**98.41**	**97.88**
S^{2}FVD$_{DPCNN}$	D2A	56.95	49.94	42.59	60.38
S^{2}FVD$_{GAT}$		57.62	58.22	58.56	57.90
S^{2}FVD$_{ERvNN}$		57.24	60.04	63.69	56.78
S^{2}FVD$_{DPCNN+GAT}$		58.20	59.70	57.68	61.88
S^{2}FVD$_{DPCNN+ERvNN}$		59.25	65.08	74.16	57.98
S^{2}FVD$_{GAT+ERvNN}$		60.11	61.27	61.61	60.93
S^{2}FVD		**63.02**	**68.99**	**76.30**	**62.95**

(b) Performance for vulnerability type detection.

Method	Dataset	Accuracy	Weighted F_1	Weighted Prec.	Weighted Rec.
S^{2}FVD$_{DPCNN}$	SARD	87.56	86.82	89.78	87.56
S^{2}FVD$_{GAT}$		88.25	88.40	88.76	88.25
S^{2}FVD$_{ERvNN}$		91.74	91.77	91.91	91.74
S^{2}FVD$_{DPCNN+GAT}$		93.31	93.31	93.40	93.31
S^{2}FVD$_{DPCNN+ERvNN}$		92.41	92.18	92.82	92.41
S^{2}FVD$_{GAT+ERvNN}$		94.78	94.79	94.86	94.78
S^{2}FVD		**97.93**	**97.94**	**97.97**	**97.93**

5. Discussion

5.1. Threats to Validity

As highlighted in previous studies [53,54], the prevalence of mislabelling in vulnerability datasets has yet to be resolved. However, in the datasets employed in our experiments, the incidence of mislabelling was relatively low, as the samples had been annotated either by security experts or through a meticulously designed differential analysis technique [50]. Therefore, as a deep-learning-based approach, S^2FVD should exhibit resistance to occasional label noise during model training, with any effect on testing performance stemming from the low-ratio noise being negligible.

Deep and machine learning models have been demonstrated to be vulnerable to adversarial attacks across multiple domains [55–57]. It is common knowledge that programmers use semantic-preserving code obfuscations or transformations to safeguard their code, which can potentially undermine the detection capability of DL-based methods, including the proposed S^2FVD. To address this issue, one potential approach is to enforce adversarial training [58] by augmenting the training data with obfuscated or transformed adversarial samples. The investigation into how S^2FVD can be affected by code obfuscations and other potential adversarial attacks, as well as the possible strategies to mitigate these effects, would be taken as one of the interesting further works.

It should be note that S^2FVD has not undergone a systematic hyper-parameter tuning process currently. Instead, either the default or commonly used empirical values for the hyper-parameters were utilized. Despite this, the evaluation results indicate that S^2FVD, trained with the current hyper-parameter settings, exhibited highly impressive vulnerability detection capability. While a systematic or exhaustive grid-search-based hyper-parameter tuning could potentially further improve S^2FVD's detection performance, such a process would require significantly more computing resources and time. As a result, we leave it for future work as well.

5.2. Limitations

As a learning-based approach, S^2FVD faces the challenge of only issuing black-box detection results. This means that, unlike the rule-based methods that furnish supplementary information that hints at possible bug-trigging paths of the detected vulnerabilities [59], it provides only a vulnerable/non-vulnerable prediction or a specific vulnerability type without explanations. As part of future work, explainable AI techniques will be combined to highlight the statements or paths with significant contributions to the prediction outcomes.

In contrast to the conventional rule-based detection methods, as well as the other deep-learning-based vulnerability detection methods that operate solely on a single code view, S2FVD's approach of extracting and combining semantic features from multiple views inevitably incurs a heavier overhead runtime. Although we acknowledge the significance of both the detection efficacy and efficiency, it is believed that the former carries greater importance in the realm of vulnerability detection. Moreover, as the computing power from both CPUs and GPUs continues to advance, achieving a moderately fast detection speed will become more feasible.

Similar to most existing works, the dataset we have primarily used is labeled at the function level. However, vulnerabilities can cross function boundaries. Therefore, designating a function as vulnerable solely because the vulnerability is revealed within it might not always be correct. Unfortunately, establishing such datasets of precisely labeled bug-triggering code contexts remains a challenging task that necessitates continuous and arduous effort from domain experts. In addition, S2FVD's assessment was limited to code written in C and C++, which are among the languages that have been hit hard by vulnerabilities. Hence, its capability in detecting vulnerabilities that transcend function boundaries, as well as its applicability to other programming languages, deserve further investigation.

6. Conclusions

Aiming at the problem that existing vulnerability detection methods that operate on a single code view are limited in detecting deep vulnerabilities, this work presents S^{2}FVD, which adopts a strategy of learning vulnerable indicative features from different code perspectives and fusing them to enhance the detection capability. In particular, to make the semantics implied within the AST be effectively encoded, an extended tree-structured neural network called ERvNN was devised. It supports the direct encoding of multi-way tree structures by implementing a GRU-style aggregation optimization for the nodes within the tree. Through the extensive experiments conducted on two large datasets consisting of both synthetic and real-world samples, a superior vulnerability detection capability of the S^{2}FVD was observed against SOTA approaches. Notably, a performance improvement of 6.86% and 14.84% regarding the accuracy and F1 metrics, respectively, was achieved for the real-world dataset D2A, thus indicating S^{2}FVD's potential in detecting vulnerabilities in more complex code contexts. Additionally, ablation studies confirmed the effectiveness of the ERvNN in encoding semantics from the AST and the superiority of the adopted multi-representation fusion strategy for boosted vulnerability detection capability.

Author Contributions: Conceptualization, Z.T.; Methodology, Z.T.; Software, B.T. and J.L.; Writing—original draft, B.T.; Writing—review & editing, J.L. and L.C.; Supervision, Z.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Natural Science Foundation of China (62272387, 61702414), the Natural Science Basic Research Program of Shaanxi (2022JM-342, 2018JQ6078), the Youth Innovation Team of Shaanxi Universities "Industial Big Data Analysis and Intelligent Processing", and the Special Funds for Construction of Key Disciplines in Universities in Shaanxi.

Data Availability Statement: The data presented in this study are available in https://github.com/lv-jiajun/S2FVD.

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

References

1. CVE. Available online: https://cve.mitre.org/ (accessed on 24 April 2023).
2. Scandariato, R.; Walden, J.; Hovsepyan, A. Predicting vulnerable software components via text mining. *IEEE Trans. Softw. Eng.* **2014**, *40*, 987–1001. [CrossRef]
3. Shaukat, K.; Luo, S.; Varadharajan, V. A novel deep learning-based approach for malware detection. *Eng. Appl. Artif. Intell.* **2023**, *122*, 106030. [CrossRef]
4. Tian, Z.; Wang, Q.; Gao, C.; Chen, L.; Wu, D. Plagiarism detection of multi-threaded programs via siamese neural networks. *IEEE Access* **2020**, *8*, 160802–160814. [CrossRef]
5. Tian, Z.; Huang, Y.; Xie, B.; Chen, Y.; Chen, L.; Wu, D. Fine-grained compiler identification with sequence-oriented neural modeling. *IEEE Access* **2021**, *9*, 49160–49175. [CrossRef]
6. Tian, Z.; Tian, J.; Wang, Z.; Chen, Y.; Xia, H.; Chen, L. Landscape estimation of solidity version usage on ethereum via version identification. *Int. J. Intell. Syst.* **2021**, *37*, 450–477. [CrossRef]
7. Russel, R.; Kim, L.; Hamilton, L. Automated vulnerability detection in source code using deep representation learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications, Orlando, FL, USA, 17–20 December 2018; pp. 757–762.
8. Li, Z.; Zou, D.; Xu, S. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
9. Zhou, Y.; Liu, S.; Siow, J. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 1–11.
10. Sun, H.; Cui, L.; Li, L. VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches. *Comput. Secur.* **2021**, *110*, 102417. [CrossRef]
11. Jang, J.; Agrawal, A.; Brumley, D. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 48–62.
12. FlawFinder. Available online: https://dwheeler.com/flawfinder/ (accessed on 10 April 2023).
13. Younis, A.; Malaiya, Y.; Anderson, C. To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, New Orleans, IL, USA, 9–11 March 2016; pp. 97–104.

14. Hin, D.; Kan, A.; Chen, H. 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.
15. Yang, S.; Cheng, L.; Zeng, Y. Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection. In Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, China, 21–24 June 2021; pp. 154–196.
16. Vadayath, J.; Eckert, M.; Zeng, K.; Weideman, N.; Menon, G.P.; Fratantonio, Y.; Balzarotti, D.; Doupé, A.; Bao, T.; Wang, R.; et al. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *31st USENIX Security Symposium (USENIX Security 22)*; USENIX Association: Boston, MA, USA, 2022; pp. 413–430.
17. Beaman, C.; Redbourne, M.; Mummery, J.D.; Hakak, S. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Comput. Secur.* **2022**, *120*, 102813. [CrossRef]
18. Zheng, P.; Zheng, Z.; Luo, X. Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*; Ser. ISSTA 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 740–751.
19. D'Silva, V.; Kroening, D.; Weissenbacher, G. A survey of automated techniques for formal software verification. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2008**, *27*, 1165–1178. [CrossRef]
20. Li,Z.; Zou, D.Q.; Xu, S.H. VulPecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16). Association for Computing Machinery, New York, NY, USA, 5–8 December 2016; pp. 201–213.
21. Cui, L.; Hao, Z.; Jiao, Y. Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Trans. Inf. Forensics Secur.* **2020**, *16*, 2004–2017. [CrossRef]
22. Li, Z. Survey on static software vulnerability detection for source code. *Chin. J. Netw. Inf. Secur.* **2019**, *5*, 1–14.
23. Kim, S.; Woo, S.; Lee, H.; Oh, H. Vuddy: A scalable approach for vulnerable code clone discovery. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 595–614.
24. Infer, Infer: A Tool to Detect Bugs in Java and c/c++/objective-c Code before It Ships. 2013. Available online: https://fbinfer.com (accessed on 12 April 2023).
25. CodeChecker. 2013. Available online: https://codechecker.readthedocs.io/en/latest (accessed on 20 April 2023).
26. Checkmarx, Checkmarx. 2022. Available online: https://www.checkmarx.com (accessed on 28 April 2023).
27. Stephan, L.; Sebastian, B.; Alexander, P. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, Republic of Korea, 18–22 July 2022; pp. 544–555.
28. Goseva-Popstojanova, K.; Perhinschi, A. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Tech.* **2015**, *68*, 18–33. [CrossRef]
29. Ghaffarian, S.M.; Shahriari, H.R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.* **2017**, *50*, 1–36. [CrossRef]
30. Perl, H.; Dechand, S.; Smith, M. 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.
31. Bosu, A.; Carver, J.C.; Hafiz, M. Identifying the characteristics of vulnerable code changes: An empirical study. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 257–268.
32. Lin, G.; Wen, S.; Han, Q.L. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
33. Li, Z.; Zou, D.; Xu, S. Vuldeepecker: A deep learning-based system for vulnerability detection. In Proceedings of the 2018 25th Annual Network and Distributed System Security Symposium (NDSS'18), San Diego, CA, USA, 18–21 February 2018; pp. 1–15.
34. Dam, H.K.; Pham, T.; Ng, S.W. A deep tree-based model for software defect prediction. *arXiv* **2018**, arXiv:1802.00921.
35. Li, Y.; Wang, S.; Nguyen, T.N. Vulnerability detection with fine-grained interpretations. In Proceedings of the 2021 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 292–303.
36. Johnson, R.; Zhang, T. Deep pyramid convolutional neural networks for text categorization. In Proceedings of the 2017 55th Annual Meeting of the Association for Computational Linguistics, Vancouver, WA, USA, 30 July–4 August 2017; pp. 562–570.
37. Mikolov, T.; Sutskever, I.; Chen, K. Distributed representations of words and phrases and their compositionality. *Adv. Neural Inf. Process. Syst.* **2013**, *26*, 3111–3119.
38. Wolf, L.; Hanani, Y.; Bar, K. Joint word2vec Networks for Bilingual Semantic Representations. *Int. J. Comput. Linguist. Appl.* **2014**, *5*, 27–42.
39. He, K.; Zhang, X.; Ren, S. Identity mappings in deep residual networks. In Proceedings of the 2016 14th European Conference of the Computer Vision–ECCV, Amsterdam, The Netherlands, 11–14 October 2016; pp. 630–645.
40. Joern. Available online: https://joern.readthedocs.io/en/latest/ (accessed on 20 April 2023).
41. Wang, X.; Ji, H.; Shi, C. Heterogeneous graph attention network. In Proceedings of the 2019 the World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 2022–2032.

42. Defferrard, M.; Bresson, X.; Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 3844–3852.
43. Baxter, I.D.; Yahin, A.; Moura, L. Clone detection using abstract syntax trees. In Proceedings of the 1998 International Conference on Software Maintenance, Bethesda, ML, USA, 16–19 March 1998; pp. 368–377.
44. Tang, Z.; Shen, X.; Li, C. AST-trans: Code summarization with efficient tree-structured attention. In Proceedings of the 2022 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; pp. 150–162.
45. Zhang, J.; Wang, X.; Zhang, H. A novel neural source code representation based on abstract syntax tree. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 783–794.
46. Pycparser. Available online: https://pypi.org/project/pycparser/ (accessed on 1 April 2023).
47. Ma, J.; Gao, W.; Wong, K.F. Rumor detection on twitter with tree-structured recursive neural networks. In Proceedings of the 2018 the Association for Computational Linguistics, Melbourne, Australia, 15–20 July 2018; pp. 1980–1986.
48. SARD. Available online: https://samate.nist.gov/SARD/ (accessed on 5 September 2022).
49. ANTLR. Available online: https://www.antlr.org/ (accessed on 5 October 2022).
50. Zheng, Y.; Pujar, S.; Lewis, B. D2A: A dataset built for ai-based vulnerability detection methods using differential analysis. In Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice, Virtual Event, Spain, 25–28 May 2021; pp. 111–120.
51. Vaswani, A.; Shazeer, N.; Parmar, N. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 6000–6010.
52. Mou, L.; Li, G.; Zhang, L. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the 2019 the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 29–32 January 2019; pp. 1287–1293.
53. Croft, R.; Babar, M.A.; Kholoosi, M. Data quality for software vulnerability datasets. In Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering (ICSE'23), Melbourne, Australia, 14–20 May 2023; pp. 1–13.
54. Jimenez, M.; Rwemalika, R.; Papadakis, M.; Sarro, F.; Traon, Y.L.; Harman, M. The importance of accounting for real-world labelling when predicting software vulnerabilities. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, San Francisco, CA, USA, 1–8 December 2023; pp. 695–705.
55. Shaukat, K.; Luo, S.; Chen, S.; Liu, D. Cyber threat detection using machine learning techniques: A performance evaluation perspective. In Proceedings of the 2020 International Conference on Cyber Warfare and Security (ICCWS), Islamabad, Pakistan, 20–21 October 2020; pp. 1–6.
56. Shaukat, K.; Luo, S.; Varadharajan, V.; Hameed, I.A.; Xu, M. A survey on machine learning techniques for cyber security in the last decade. *IEEE Access* **2020**, *8*, 222310–222354. [CrossRef]
57. Shaukat, K.; Luo, S.; Varadharajan, V.; Hameed, I.A.; Chen, S.; Liu, D.; Li, J. Performance comparison and current challenges of using machine learning techniques in cybersecurity. *Energies* **2020**, *13*, 2509. [CrossRef]
58. Shaukat, K.; Luo, S.; Varadharajan, V. A novel method for improving the robustness of deep learning-based malware detectors against adversarial attacks. *Eng. Appl. Artif. Intell.* **2022**, *116*, 105461. [CrossRef]
59. Cheng, X.; Nie, X.; Li, N.; Wang, H.; Zheng, Z.; Sui, Y. How about bug-triggering paths?—Understanding and characterizing learning-based vulnerability detectors. *IEEE Trans. Dependable Secur. Comput.* **2022**, 1–18. [CrossRef]