


Article

Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning

Xin Li ^{1,2,3}, Lu Wang ¹ , Yang Xin ^{1,2,3,*}, Yixian Yang ^{1,2,3} and Yuling Chen ²

¹ School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China; li_xin@bupt.edu.cn (X.L.); wltongxue@bupt.edu.cn (L.W.); yxyang@bupt.edu.cn (Y.Y.)

² Guizhou Provincial Key Laboratory of Public Big Data, Guizhou University, Guiyang 550025, China; ylchen3@gzu.edu.cn

³ National Engineering Laboratory for Disaster Backup Recovery, Beijing 100876, China

* Correspondence: yangxin@bupt.edu.cn

Received: 9 January 2020; Accepted: 26 February 2020; Published: 2 March 2020



Abstract: Vulnerability is one of the root causes of network intrusion. An effective way to mitigate security threats is to discover and patch vulnerabilities before an attack. Traditional vulnerability detection methods rely on manual participation and incur a high false positive rate. The intelligent vulnerability detection methods suffer from the problems of long-term dependence, out of vocabulary, coarse detection granularity and lack of vulnerable samples. This paper proposes an automated and intelligent vulnerability detection method in source code based on the minimum intermediate representation learning. First, the sample in the form of source code is transformed into a minimum intermediate representation to exclude the irrelevant items and reduce the length of the dependency. Next, the intermediate representation is transformed into a real value vector through pre-training on an extended corpus, and the structure and semantic information are retained. Then, the vector is fed to three concatenated convolutional neural networks to obtain high-level features of vulnerability. Last, a classifier is trained using the learned features. To validate this vulnerability detection method, an experiment was performed. The empirical results confirmed that compared with the traditional methods and the state-of-the-art intelligent methods, our method has a better performance with fine granularity.

Keywords: cyber security; vulnerability detection; program slice; transfer learning; representation learning

1. Introduction

The importance of cyberspace security has become more and more significant. However, cyberspace is facing a serious threat of invasion. The root cause of most cyber-attacks is vulnerabilities. Vulnerabilities exploited by attackers compromise the confidentiality, integrity, and availability of information systems. For instance, the ransomware Wannacry which exploited a vulnerability in Windows server message block protocol has swept the world [1]. Early detection of vulnerability is an effective way to reduce the loss. Despite the efforts of experts and scholars, vulnerabilities remain a huge problem and will continue to exist in the long term. This can be justified by the fact that an increasing number of vulnerabilities are published every year [2].

Vulnerability detection is a method to discover vulnerabilities in software. Traditional vulnerability detection includes static and dynamic methods [3]. Static methods, such as data flow analysis [4,5], symbol execution [6], and theorem proving [7], analyze source code or executable code without running software. Static methods have high coverage and can be deployed in the early stage of

software development. However, it suffers from a high false positive rate. Dynamic methods, such as fuzzy test [8] and dynamic symbol execution [9], verify or discover the nature of software by running the program. Dynamic methods have a low false positive rate and simple deployment, but their dependence on the coverage of test cases incurs a low recall. Therefore, realizing automatic and intelligent vulnerability detection is the trend of research. This can be justified by the organization of DARPA's Cyber Grand Challenge [10].

The development of machine learning technology provides new methods to alleviate the bottlenecks of traditional methods. Intelligent vulnerability detection methods which operate on source code are one of the main research directions. It can be categorized into 3 types: using software engineering metrics, anomaly detection, and vulnerable pattern learning [11]. In the early stages, software engineering metrics, such as software complexity [12], developer activities [13], and code commits [14] are explored to train a machine learning model. The inspiration for this approach is that the more complex the software is, the more vulnerable it is. Software engineering metrics methods have the advantages of speediness and easily acquired datasets, but its effectiveness in accuracy and recall is still to be improved. Anomaly detection and vulnerable pattern learning both try to improve the detection effect by utilizing the syntactic and semantic information in the code [15]. Anomaly detection learns the legal programming pattern from mature software codes. The similarity or association between the candidates and the learned rules is used to detect vulnerabilities [16]. Anomaly detection has the advantage of discovering unknown vulnerabilities. However, the false positive and false negative is high. The vulnerable pattern learning method learns vulnerable patterns from vulnerable and clean samples [17]. Compared with anomaly detection, this method has a better performance on the accuracy, but highly dependent on the quality of the dataset.

The intelligent vulnerability detection methods leverage software syntax and semantic information to improve detection performance, but there are several problems that compromise the effect of existing intelligent methods: (1) long-term dependency between code elements. In a vulnerable sample, the dependency between context can be long. For example, variables defined at the beginning of a program may be used at the end. This may cause the machine learning algorithm to ignore the correlation between context when detecting the vulnerability; (2) out-of-vocabulary (OoV) issue. Program language allows users to customize identifiers, such as variable name, function name, and class name, and every programmer has his own style when naming identifiers. This leads to the lack of a common vocabulary to cover all possible identifiers. Hence, there is always an out-of-vocabulary issue during vulnerability detection. This will weaken the effect of vulnerability detection; (3) coarse detection granularity. A coarse detection granularity, such as a component, function or file, cannot properly represent vulnerability information. For example, a vulnerability might cross the boundary of a function or a file. A coarse detection granularity may incur noise when detecting vulnerabilities. In addition, coarse detection granularity provides imprecise information to assist developers to patch. Thus a considerable human effort is still required to pinpoint the vulnerable code fragments; (4) lack of vulnerability dataset. The quantity and quality of training data determine the effectiveness of intelligent vulnerability detection methods. However, due to the particularity of vulnerability, the lack of a general and authoritative vulnerability dataset for training and testing still limits the performance of intelligent methods.

To overcome these challenges, we proposed a framework that detects software vulnerabilities in four stages: pre-processing, pre-training, representation learning, and classifier training. In the pre-processing stage, our method transforms the samples in the form of raw source code into the minimum intermediate representations through dependency analysis, program slicing, tokenization, and serialization. The length of dependencies between context is reduced by eliminating irrelevant code. The samples used in the next three stages are pre-processed respectively. In the pre-training stage, considering the lack of vulnerability samples, we conduct unsupervised learning on an extended corpus. The purpose of this process is to learn the common syntax features of program language and alleviate the OoV issue through distributed embedding. The result of the pre-training stage will

serve as the parameters of the embedding layer in the next two stages. In the representation learning stage, three concatenated convolutional neural networks are utilized to obtain high-level features from a vulnerability dataset. In order to train this network, two dense layers are added in this stage. In the classifier training stage, the vulnerability dataset is transformed into high-level features using the learned network in the last stage, and a classifier is trained for vulnerability detection by the learned high-level features. Finally, test samples will be classified by the trained model. The empirical study shows that compared with the traditional methods and state-of-the-art intelligent methods, our method has made significant improvements in false positive rate, false negative rate, precision, recall, and F1 metrics.

The remainder of this paper is organized as follows: We introduce the related work in Section 2. In Section 3 motivating examples and hypotheses are introduced. Our method for automated vulnerability detection in source code is presented in Section 4. We introduce the experiments and discuss the results in Section 5. Finally, we conclude this paper in Section 6.

2. Related Work

2.1. Intelligent Vulnerability Detection

To detect vulnerability automatically, researchers have proposed several approaches. Fabian [18] proposed an anomaly detection method for taint-style vulnerabilities. It clusters the initialization of variables that can be propagated to security-sensitive functions. Then, the violation is reported as potential vulnerabilities by anomaly detection. This approach is suitable for taint-style vulnerability but not for universal vulnerability. Kim [19] proposed a vulnerability detection method based on similarity. However, this method is limited to vulnerabilities caused by code cloning. Wang [20] proposed a representation learning method with file-level granularity. It extracts three types of nodes from the abstract syntax tree (AST) of a file: declaration, control-flow and invocation. A deep belief network (DBN) is used to learn advanced features. The information extracted by this method is too coarse to detect vulnerabilities caused by improper use of variables. Lin [21,22] and Michael [23] proposed an AST-based intra-procedural representation learning method to detect vulnerabilities. However, they both suffer from coarse detection granularity. In addition, their performances are compromised by the presence of irrelevant codes. Bian [24] proposed an anomaly detection method based on static analysis. He converts the program slice to an AST and uses a hash algorithm to encode the AST. Li [25,26] proposed a detection method that learns vulnerable programming patterns with token granularity. A classifier is trained using the embedded samples directly without representation learning. Furthermore, its embedding and classification model are learned from the same dataset. This limits its performance. To solve the problem of information loss in the process of representation learning, Zhou [27] proposed a method that used the graph neural network for vulnerability detection with function-level granularity. The samples are converted into the form of code property graph. Then a graph neural network that is composed of a gated graph recurrent layer and a convolutional layer is trained to learn the vulnerable programming pattern. This approach improves the detection of intra-procedural vulnerabilities. However, it fails to cover inter-procedural vulnerabilities.

2.2. Program Understanding Model

The program understanding model is the basis of intelligent vulnerability detection. There are two main forms of program understanding: sequence and structure. The sequential understanding model converts the source code into a sequence in a certain order, including character [28], token [29] and API [30]. It retains native information. However, it is affected by long-term dependency. The structural program understanding model includes Abstract Syntax Tree (AST) [24,31], Control Flow Graph (CFG) [32], Program Dependence Graph (PDG) [33], and Code Property Graphs (CPG) [34]. AST represents the syntactic structure of a programming language in a tree form. Through an AST, the syntax information can be obtained. CFG takes the codes with a sequential relationship as a basic block

and concatenates them into an ordered graph based on their control dependency. CFG specifies the execution sequence of statements and the required conditions for a particular execution sequence. However, CFG failed to identify data flow information. PDG adds data dependency and control dependency on the nodes of CFG. The advantage of this structure is that statements that affect some sensitive operations can be easily and accurately identified. However, PDG loses the syntax information which is crucial for the detection of some types of vulnerability. Given that AST, CFG, and PDG have their own priorities, Yamaguchi [34] proposed CPG that combines them into one graph. Although CPG provides accurate and detailed information, it compromises the efficiency of detection due to the increased data to be analyzed. Furthermore, structural representation is more complicated than sequential representation. In order to balance accuracy and efficiency, some intelligent vulnerability detection methods [25,26] transform natural code sequence into a structured model. Then the structure model is transformed into a sequence model before embedding.

3. Preliminary

3.1. Motivating Examples

Vulnerability is a defect or fault that can be exploited by a malicious user to make the software perform operations different from its design logic. From a source code perspective, most vulnerabilities are rooted in a critical operation that raises security issues. The critical operation can be a function, an assignment or a control statement. An attacker can directly or indirectly influence this critical operation by controlling certain variables or conditions.

Buffer overflow vulnerability (CWE-119) is selected to illustrate the feature of vulnerability. This is because the buffer overflow vulnerability has become the most impacted vulnerability, according to a report from Common Weakness Enumeration in 2019 [35]. In addition, it is representative because of complex data dependence and control dependence. Although buffer overflow vulnerability is selected as an example, our method is universal for other types of vulnerability detection.

Buffer overflow refers to the writing of data in the buffer that exceeds its own length, resulting in the overwriting of storage units outside the buffer. Figure 1 illustrates an intra-procedural buffer overflow vulnerability in file “example_1.c.” The variable “buf” is allocated a buffer of 10 bytes and assigned values through a “for” loop. When the loop reaches the eleventh times ($i = 10$), “buf” can still be assigned. This operation overflows the buffer of “buf.” In this sample, all the relevant codes for the vulnerability are inside the procedure “main.” The critical operation is an assignment in line 12. It is data-dependent on line 5 and control dependent on lines {4, 10}. The scope of this vulnerability is lines {4, 5, 10, 12}. In this example, the variable “buf” is declared on line 6 but assigned on line 13. This reflects the long-term dependence issue. In addition, lines {1, 2, 3, 6, 7, 8, 9, 11, 13, 14} are irrelevant statements. These factors will interfere with the learning of vulnerable programming patterns. In the real-world programs, the percentage of irrelevant code may be high.

```

1 void bar(char *buf, char *src) {
2     strcpy(buf, src);
3 }
4 int main() {
5     char buf[10];
6     char src[10];
7     memset(src, 'A', 10);
8     src[10 - 1] = '\0';
9     bar(buf, src);
10    for (int i = 0; i <= 10; i++)
11        //writes buf [10] and overruns memory
12        buf[i] = 'B';
13    return 0;
14 }

```

Figure 1. File “example_1.c” with an intra-procedural buffer overflow vulnerability.

Figure 2 illustrates an inter-procedural buffer overflow vulnerability in file “example_2.c.” In the function “test,” the variable “buf” is allocated a buffer of 64 bytes. However, it can be copied

to 1024 bytes data at most in line 6. When the size of the variable “str” exceeds 64 bytes, it will overflow the buffer of “buf”. The value of the variable “str” comes from the main function. This is an inter-procedural vulnerability whose critical operation is a data copy in line 6. The scope of this vulnerability is lines {12, 13, 15, 16, 17, 1, 3, 4, 6}.

```

1 void test(char *str){
2     char *buf;
3     buf = malloc(64);
4     if(!buf)
5         return;
6     snprintf(buf, 1024, "<%s>",str);
7     printf("result: %s\n", buf);
8     free(buf);
9 }

12 int main(int argc, char **argv){
13     char *userstr;
14
15     if(argc > 1) {
16         userstr = argv[1];
17         test(userstr);
18     }
19     return 0;
20 }

```

Figure 2. File “example_2.c” with an inter-procedural buffer overflow vulnerability.

As can be seen from the above two examples, many instructions are irrelevant to a vulnerability. Therefore, detection with function-level or file-level granularity will be interfered with by the irrelevant instructions. The second example illustrates that vulnerability may span multiple functions. Therefore, a method with improper detection granularity fails to discover the inter-procedural vulnerabilities. In addition, function-level and file-level granularity is too coarse to assist the developer with an audit. A lot of human intervention is needed to pinpoint the vulnerabilities and make fixes.

3.2. Hypothesis

Programming language can be regarded as a language of communication between human beings and computers. It has many similarities with natural language. For example, they are both composed of tokens and can be resolved into a syntax tree. Success in the field of natural language processing enlightens us to borrow concepts from natural language processing (NLP) for vulnerability detection.

In natural language processing, a sentence is represented as a sequence of the token. The context of a word is its preceding and succeeding words. Therefore, distributed representations are based on an assumption: Words that occur in the same context tend to have similar meanings [36].

In vulnerability detection, code can be converted into a sequence by some rules, such as dependency or sequence of program execution. After the transformation, it has the same form as a natural language. Therefore, we make assumptions for vulnerability detection:

Hypothesis 1. *In a programming language, the context of a token is its preceding and succeeding tokens, and tokens that occur in the same context tend to have similar semantics.*

Hypothesis 2. *The same types of vulnerabilities have common semantic characteristics. These characteristics can be learned from the context of vulnerabilities.*

4. Proposed Approach

4.1. Overview

Our study aims to automatically detect vulnerabilities in the software while providing precise information to assist developers to audit. Our method selects an inter-procedural slice as the detection granularity. Figure 3 shows an overview of our proposed method. The black arrow represents the data flow during the pre-training stage. The red arrow represents the data flow during the representation learning stage. The blue arrow represents the data flow during the classifier training stage. Although the arrows in the pre-processing phase are black, the data in the other three stages will flow through this stage. As shown in Figure 3, the inputs are in the form of source code. In the pre-processing stage, through dependency analysis, security slicing, tokenization, and serialization, we obtain a sequential minimum intermediate representation of the sample. In the pre-training stage, we learn a

distributed embedding from an extended corpus. The output of this stage is used as the parameters of the embedding layer in the next two stages. In the representation learning stage, three concatenated convolutional neural networks are used to learn high-level features. The trained model is reused in the next stage. In the classifier training stage, a machine learning classifier is trained using the outputs of the convolutional layer. Finally, test samples will be detected by the trained model in the last stage to predict whether they are vulnerable or not.

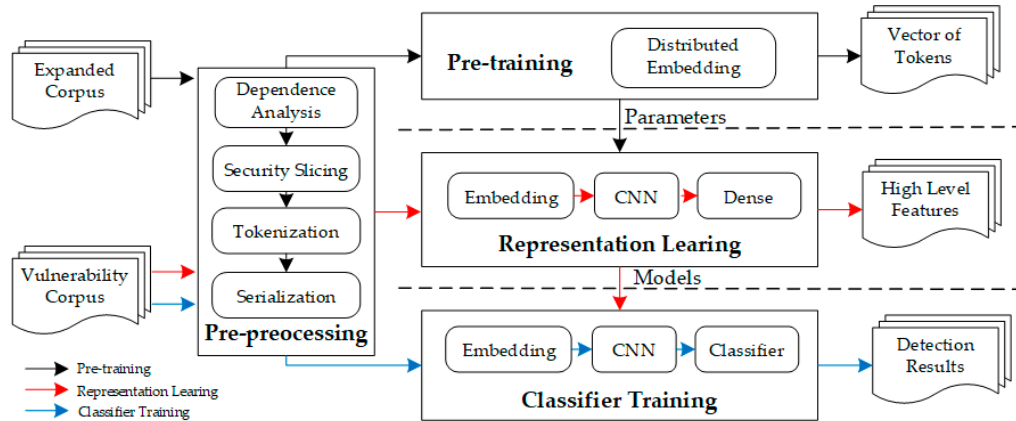


Figure 3. Overview of the proposed intelligent vulnerability detection method.

4.2. Pre-Processing

A fine-granularity detection method can not only eliminate extraneous features in the analysis process but also provide detailed information to audit. Different from file-level and function-level granularity in many papers, our method chooses the **subset of code related** to vulnerabilities as detection granularity. We define minimum intermediate representation (MIR) as a sequence of tokens which are serialized from an **inter-procedural program slice** that takes a security-critical operation as a criterion. The order between tokens represents dependencies.

As shown in Figure 4, the samples in the form of the source code are converted to minimum intermediate representations by static analysis in the pre-processing stage. First samples in the form of source code are converted into program dependency graphs by control dependency and data dependency analysis.

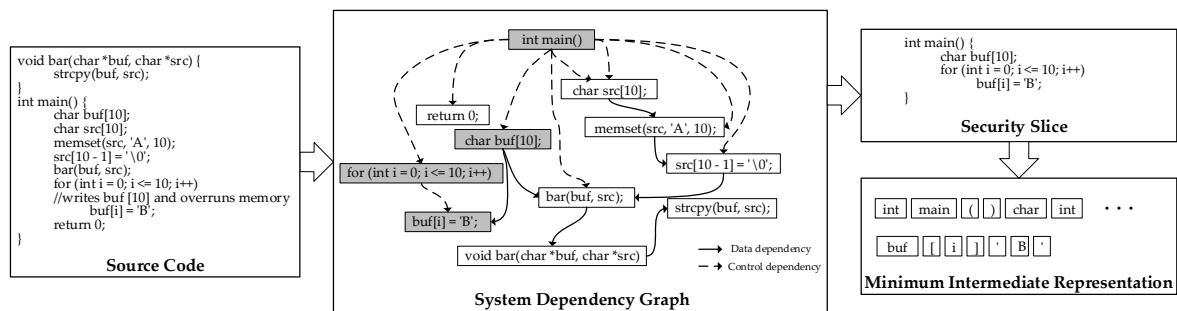


Figure 4. Overview of the pre-training stage. The solid arrows denote data dependencies and dashed arrows denote control dependencies in the system dependency graph.

Control dependency between statement i and statement j satisfies that $\exists path : [i, j]$, (1) j post-dominate k , $k \in (i, j)$; (2) i is not post-dominate by j .

Data dependency between statement i and statement j satisfies that $\exists x \in V$, (1) $x \in DEF[i]$ and $x \in USE[j]$; (2) $\forall k \in (i, j)$, $x \notin DEF[k]$. V denotes the set of variables in a program. $DEF[i]$ denotes the set of variables defined at statement i , and $USE[j]$ denote the set of variables referenced at statement j .

In this paper, a program dependence graph (PDG) is a directed graph $G = \langle S, E \rangle$. S is the set of the vertex in the program dependency graph that denotes the statements. The E is the set of edges that represents the control dependency and data dependency. Given many vulnerabilities are inter-procedural, we generated the system dependence graph (SDG) by adding the invocation between procedures to the program dependency graph.

Next, all the critical operations in the system dependency graph are located according to the characteristics of the vulnerability to be detected. A backward program slice is conducted on every critical operation. Then, we replace the call and declaration statements of the custom functions with a data dependency between the arguments to the parameters and a data dependency between the return values to the call points. We serialize the results of slicing in the order the code executes. So far, the basic unit is still a statement. Therefore, lexical analysis is performed to turn the slice into tokens. In the example shown in Figure 4, "buf[i] = 'B'" is a critical operation for buffer overflow vulnerability. This is because its misuse can cause data to be written out of the buffer. The relevant statements are pinpointed by dependency analysis in the SDG. They are statements {"for (int i = 0; i <= 10; i++)", "char buf[10];", "int main()". We serialize the critical operation and its associated statements in the executed order. Finally, by a custom lexical analysis, we obtain the MIR: {"int", "main", "(", ")", "{", "char", "buf", "[", "10", "]", ":", "for", "(", "int", "i", "=", "0", ":", "i", "<=", "10", ":", "i", "++", ")", "buf", "[", "i", "]", "=", "B", ""}. As a result, the source code is transformed into a form similar to natural language.

4.3. Pre-Training

Machine learning models take real value vectors as inputs. Therefore the minimum intermediate representations should be mapped to real value vectors. Based on Hypothesis 1, we distributedly represent each token with its context. Compared with traditional encoding methods, such as one-hot, term frequency-inverse document frequency (TF-IDF), n-gram, distributed representation is denser. Semantically similar identifiers will be mapped to similar vector representations due to their similar context.

In the pre-training stage, the Continuous Bag-of-Words (CBOW) model is leveraged to obtain distributed vector representations. As illustrated in Figure 5, the CBOW model is a simplified neural network with three layers. The input of the CBOW model is the vector representation of context. Therefore, the size of the input layer is $C \times N$, where C is the size of the context and N is the dimension of vector representation. The projection layer summates the vector representations, and the output is:

$$x_i = \frac{1}{C}(v_1 + v_2 + \cdots + v_C), \quad (1)$$

where v_i is the initial vector representation of token t_i . The output layer is a Huffman tree, whose leaf nodes represent the tokens in the vocabulary. Therefore, the number of leaf nodes is V and the number of the inner nodes is $V - 1$. In a Huffman tree, there is a unique path P_i from the root node to the target leaf node t_i . In the path, the j -th node is represented as $n_{i,j}$. $\theta_{i,j}$ represents the vector of $n_{i,j}$, and $d_{i,j} \in \{0, 1\}$ denotes the encoding value of the node $n_{i,j+1}$. The probability of going left or right at the inner node $n_{i,j}$ is:

$$p(d_{i,j}|x_i, \theta_{i,j-1}) = [\sigma(x_i^T \theta_{i,j-1})]^{1-d_{i,j}} \cdot [1 - \sigma(x_i^T \theta_{i,j-1})]^{d_{i,j}}, \quad (2)$$

Therefore, in the CBOW model, the probability of target token t_i appearing in a context is:

$$p(t_i | \text{context}(t_i)) = \prod_{j=2}^{l_i} p(d_{i,j} | x_t, \theta_{i,j-1}), \quad (3)$$

where $context(t_i)$ denotes the context of the target token t_i . Then, the objective function is:

$$L = \sum_{t_i \in C} \log(p(t_i | context(t_i))) = \sum_{t_i \in C} \sum_{j=2}^{l_i} \left\{ (1 - d_{i,j-1}) \cdot \log(\sigma(x_i^T \cdot \theta_{i,j-1})) - d_{i,j} \cdot \log(1 - \sigma(x_i^T \cdot \theta_{i,j-1})) \right\}, \quad (4)$$

This is an unsupervised learning method that enables us to learn programming patterns from an unlabeled dataset. Therefore, we utilize a transfer learning method to learn distributed embedding from an extended dataset. Tool word2vec [37] is used to implement the CBOW language model. The learned vector representations are used as the parameters of the embedding layer in the representation learning stage and classifier training stage.

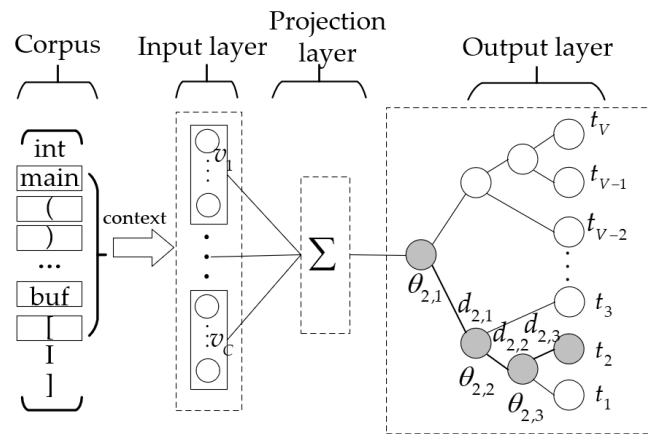


Figure 5. An example of a continuous bag-of-words model. The output layer is a Huffman tree with V leaf nodes and $V-1$ inner nodes. The path P_2 from the root node to t_2 is highlighted as an example, where $d_{2,j} \in \{0, 1\}$ denotes the encoding value of the j -th node in the path P_2 . $\theta_{2,j}$ denotes the vector of the j -th node in the path P_2 .

4.4. High-Level Feature Learning

According to Hypothesis 2, the vulnerable programming pattern of a sample is hidden in the context. We leverage a representation learning method to obtain high-level features from the serialized tokens. Convolutional neural networks have been successfully used in many tasks such as computer vision and natural language processing. It proves the ability of the convolutional neural network to learn the spatial structure in input data.

Inspired by the n-gram model in natural language processing, we use three parallel 1-dimensional convolutional neural networks to extract different features. As shown in Figure 6, the representation learning stage consists of five layers. The embedding layer map for the MIR is obtained through static analysis to a form of vector. Let $v_i \in R^k$ be the vector representation corresponding to the i -th token in a MIR. A MIR $x_j \in R^{n \times k}$ of length n represented as

$$x_j = v_1 \oplus v_2 \oplus \dots \oplus v_n, \quad (5)$$

where j is the order of the sample in the dataset and \oplus denotes the concatenate operator. In the convolution layer, let $w \in R^{h_m \times k}$ be the filter, and the window of the filter is h_m , $m \in [1, 3]$. A new feature c_i is generated by

$$c_i = f(w \cdot x_{i:i+h-1} + b), \quad (6)$$

where $b \in R$ is the bias, and f is an activation function. $x_{i:i+h-1}$ denotes the value involved in the convolution. There are d filters for each type. Therefore, the output of each convolution operation with a window h_m is $u_{j,h_m} \in R^{n \times d}$. After performing a concatenation operation on u_{j,h_m} , all features are mapped to a 2-dimensional tensor $s_j \in R^{3n \times d}$. The global-max-pooling operation is conducted on the second dimension of s_j . This will pick out the most important feature while retaining the structure

information. As a result, we obtain a high-level feature $z_j \in R^k$. In order to complete the network, two dense layers are added after the global max pooling layer, and the dense layers have k and two nodes, respectively. The convolutional layer is reused in the classifier training phase.

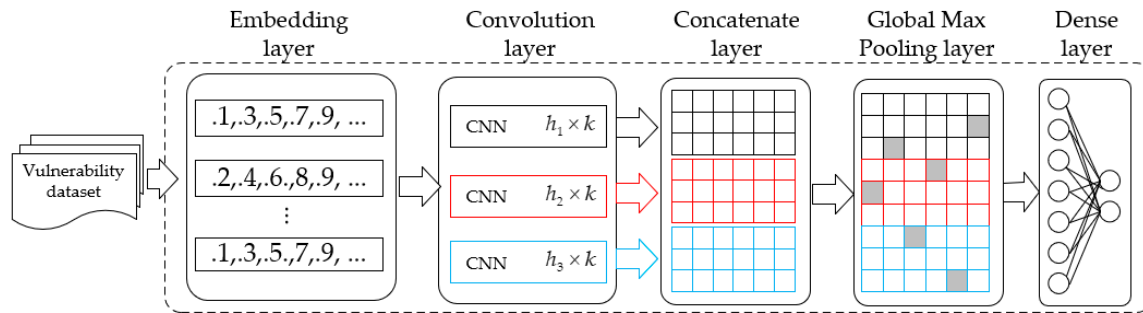


Figure 6. The overview of the representation learning stage. In the Embedding layer, the weights come from the pre-training stage. Three convolutional neural networks constitute the Convolutional layer. In this paper, the filter size of each network is h_1, h_2, h_3 respectively. The outputs of the Convolutional layer are concatenated through the Concatenate layer. A global-max-pooling operation is conducted to find the max value in every column. Lastly, two dense layers are added, and the activation function is softmax.

4.5. Building Models and Performing Vulnerability Detection

We obtain the high-level features of the vulnerability dataset by reusing the trained convolutional neural networks. The high-level features are utilized to train a classifier for predicting whether a sample is vulnerable or not.

Finally, we get the vulnerability detection model. It leverages the pre-processing method in Section 4.3 to obtain the minimum intermediate representations of the test samples. The word vectors obtained in Section 4.4 are used as parameters of the embedding layer. The convolutional neural networks trained in the representation learning stage of Section 4.5 are used to obtain high-level features. The classifier trained in the last stage is used to predict whether the test samples are vulnerable or not.

5. Experiments and Results

5.1. Evaluation Metrics

Let True Positive (TP) denote the number of correctly classified vulnerable samples, False Positive (FP) denotes the number of falsely classified clean samples, False Negative (FN) denotes the number of falsely classified vulnerable samples, and True Negative (TN) denotes the number of correctly classified clean samples. To measure vulnerability detection results, five metrics were used. False Positive Rate (FPR) = $FP / (FP + TN)$ denotes the proportion of falsely classified clean samples in all clean samples. False Negative Rate (FNR) = $FN / (FN + TP)$, denotes the proportion of falsely classified vulnerable samples in all vulnerable samples. Precision (P) = $TP / (TP + FP)$, denotes the proportion of correctly classified vulnerable samples in all samples that are classified as vulnerable. Recall (R) = $TP / (TP + FN)$, represents the ability of a classifier to discover vulnerabilities from all vulnerable samples. F1-Measure (F1) = $2 \cdot P \cdot R / (P + R)$ measure the ability of both Precision and Recall. In the experimental results, the classifiers with low FPR, FNR and high P, R and F1 metrics have excellent performance.

5.2. Experimental Setup

The dataset used in this paper comes from the Software Assurance Reference Dataset (SARD) [38] and the National Vulnerability Database (NVD) [39]. SARD comes from the Software Assurance Metrics And Tool Evaluation (SAMATE) project of the National Institute of Standards and Technology

(NIST). SARD aims to serve as a standard dataset to evaluate the effectiveness of vulnerability detection tools with known software security errors and fixes for them. The samples in the dataset come in two forms: artificially designed vulnerabilities, and vulnerabilities found in software products. The NVD is the U.S. government repository that collects vulnerabilities and their patches in software products. We choose the buffer overflow vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399) on SARD and NVD as the samples for learning vulnerable programming patterns. Other samples in C language on SARD were selected to extend the dataset for pre-training. Statistics on training data and pre-training data are summarized in Table 1. These datasets have been preliminarily processed by [25]. We set train data:test data = 7:3, and used 10-fold cross validation to choose super parameters. In our experiment, buffer overflow vulnerability was selected as the main detection object (except Section 5.5). This is because the buffer overflow vulnerability is the most significant vulnerability [35]. Therefore, the detection of this type of vulnerability has practical significance. In addition, abundant samples of the buffer overflow vulnerability are available. It should be noted that our approach also applies to other vulnerabilities caused by the misuse of data processing and control logic.

Table 1. Statistics on training data and pre-training data.

Dataset	Samples	Vulnerable	Not Vulnerable	Vocabulary
CWE-119	39,753	10,440	29,313	16,983
CWE-399	21,885	7285	14,600	7811
Extended dataset	482,265	-	-	61,638

The neural networks are implemented by Keras (version 2.2.5). The Random Forest, Gradient Boosting Decision Tree, SVM, Logistic Regression and Naive Bayesian algorithm were provided by the Scikit-learn (version 0.21.3). The distributed embedding is implemented by Word2vec provided by genism (version 3.0.1). Our algorithm was run on Google Colaboratory with Tesla T4 GPU.

In the representation learning stage, the parameters we used are shown in Table 2.

Table 2. Tuned Parameters for representation learning.

Parameter	Description
Input_dim	The size of vocabulary (16983).
Output_dim	The dimensionality of vectors that the tokens are converted to (200).
Sequence_length	The length of each sample (400).
CNN units	There are 3 concatenated convolutional neural networks (CNN). The number of filters all is 128. The size of filters is 3, 5, 7 respectively.
Batch_size	The number of samples that are propagated through the network (128).
Loss function	A function to calculate the loss between the predicted value and real value (binary_crossentropy)
Optimizer	The algorithm to optimize the neural network (Adam)
Monitor	The metric to be monitored for early stop (F1) and patience (10)

5.3. Comparison of Different Neural Networks

In order to select the best model for representation learning, we adopted a 10-fold cross validation on train data. We compared six neural networks. Sequential CNN has three sequentially connected convolutional neural networks (CNN). Each convolutional neural network has 128 filters, and the size of filters is 3, 5, 7, respectively. Sequential LSTM has two long short-term memory neural networks (LSTM). Each LSTM has a 128-dimensional output. BiLSTM is composed of a forward LSTM and a backward LSTM, and it has a 128-dimensional output. In CNN + BiLSTM, the output of CNN with 128

5-dimensional filters is the input of LSTM with 128 output. CNN + BiLSTM + Attention has an extra self-attention layer in addition to CNN + BiLSTM. Concatenated CNN has an output that concatenates the output of three convolutional neural networks. Each neural network is connected to a dense layer for training. The comparison results are showed in Table 3.

Table 3. Comparison of different representation learning models.

Model	FPR (%)	FNR (%)	P (%)	R (%)	F1 (%)
Sequential CNN	6.3	22.0	85.9	68.0	75.9
Sequential LSTM	17.6	30.8	69.0	69.2	69.1
BiLSTM	29.4	7.3	64.1	92.3	75.7
CNN+BiLSTM	13.1	10.4	80.6	89.6	84.9
CNN +BiLSTM+Attention	5.5	33.9	87.1	66.1	75.1
Concatenated CNN	1.8	15.0	94.4	85.0	89.5

We observe that although BiLSTM achieves the best results on FNR and Recall, it has the worst results on P. Compared with other models, Concatenated CNN has balanced performance and achieves the best results in FPR, P, and F1. Therefore, in the representation learning stage, we chose Concatenated CNN as the final model to learn high-level features.

In order to compare the effects of different classifiers at the stage of vulnerability detection, we used the learned high-level features to train six classifiers: Logistic Regression (LR), Naive Bayesian (NB), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), Gradient Boosting Decision Tree (GBDT), Random Forest (RF). Table 4 shows the comparison results. Among the six classifiers, SVM had the best performance on FPR and P metrics, while RF achieved the best performance on FNR, R, and F1 metrics. Therefore, RF has a more balanced performance.

Table 4. Comparison of different classification algorithms.

Algorithm	FPR (%)	FNR (%)	P (%)	R (%)	F1 (%)
LR	2.4	12.0	93.0	88.0	90.5
NB	18.4	13.1	62.6	86.9	72.8
SVM	1.2	10.9	96.4	89.1	92.6
MLP	2.8	12.6	91.8	87.4	89.5
GBDT	1.3	11.3	96.0	88.7	92.2
RF	1.5	9.6	95.7	90.4	93.0

5.4. Effectiveness of Pre-Training

In order to validate the effectiveness of pre-training, we projected the 200-dimensional embeddings to 2-dimensional vectors. Three types of tokens with different semantics were picked out. They are memory manipulation functions, logical comparisons and custom identifiers. The projected embeddings are shown in Figure 7.

As can be seen from Figure 7, different types of tokens were grouped into separable clusters. This proves that through the pre-training on the extended corpus, our method can learn the common semantic information. In addition, custom identifiers with similar semantics have similar positions. This proves the ability of our method to alleviate the OoV problem.

In addition, we compared the effectiveness of our method with four different embedding methods. A random forest is used as a classifier in all five methods. The first three methods that use Vocabulary, N-Gram, IT-IDF language models respectively were performed on the same dataset (CWE-119). Their classifiers are trained using the real value vectors encoded respectively. The last two methods both use the CBOW language model as the embedding method, and a representation learning is performed to obtained high-level features. The difference is that the former uses the same dataset (CWE-119) for pre-training, representation learning, and classifier training. The latter is our approach that utilizes an

extended corpus for pre-training and vulnerability dataset (CWE-119) for representation learning and classifier training.

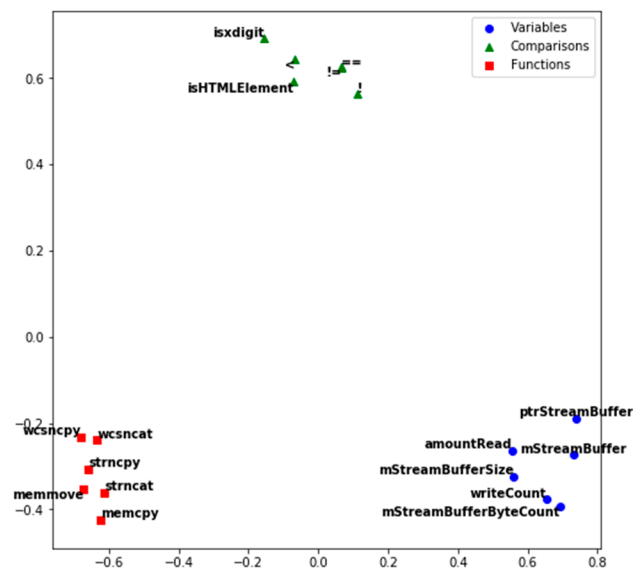


Figure 7. The embedding result of three types of tokens. The blue dots denote custom identifiers. The green triangles denote logical comparisons. The red quadrates denote memory manipulation functions.

Table 5 summarizes the experimental results. The first three methods have approximate performances. A significant improvement over the first three methods is achieved by the fourth method. This shows that the application of pre-training can improve the effectiveness of vulnerability detection. The last result is achieved by our method that has the best performance. This is because, through pre-training from a large database, the common syntax of a programming language is learned, and every token is represented by its context. Even if tokens are unique in a different program, it will eventually be represented as a similar vector as long as the identifier has a similar context. By using this transfer learning approach, we mitigate the impact of the OoV issue. The results validate Hypothesis 1.

Table 5. Comparison of five different embedding methods.

Method	Corpus	FPR (%)	FNR (%)	P (%)	R (%)	F1 (%)
Vocabulary	CWE-119	4.9	31.6	82.8	68.4	75.0
N-Gram	CWE-119	6.7	27.2	79.1	72.8	75.8
TF-IDF	CWE-119	6.2	27.2	80.4	72.8	76.4
CBOW	CWE-119	3.1	20.4	89.9	79.6	84.4
CBOW	Extended corpus	1.5	9.6	95.7	90.4	93.0

5.5. Ability to Detect Different Vulnerabilities

In order to verify the detection ability of our method for different types of vulnerability, we conduct experiments on two datasets: CWE-119 and CWE-399. CWE-119 was the dataset of buffer overflow vulnerability. CWE-399 was the dataset of resource management error vulnerability. The two experiments were performed with the same super parameters and experimental procedures. The results are summarized in Table 6.

As shown in Table 6, our proposed method carried out effective detection on both two datasets. This validates that our detection method can be applied to different types of vulnerability. In comparison, the detection effect of CWE-399 was better than that of CWE-119. This is because buffer overflow vulnerability has more complex forms than resource management error vulnerability, such as complex

data and control dependencies and various sanitization methods. As a result, this makes it difficult for machine learning algorithms to learn vulnerability patterns from samples.

Table 6. Performance of our method on different types of vulnerability.

Vulnerability	FPR (%)	FNR (%)	P (%)	R (%)	F1 (%)
CWE-119	1.5	9.6	95.7	90.4	93.0
CWE-399	0.1	3.0	98.7	97.0	97.8

The execution time and memory space of our proposed method are summarized in Table 7. In the pre-training stage, our method took 216.5 s and used 278.0 MB of memory space. In the representation learning stage, the execution time of CWE-119 was 301.1 s and the execution time of CWE-399 was 137.4 s. The memory space was 2356.6 MB and 2341.9 MB, respectively. In the classifier training stage, execution time was 7.0 s and 4.2 s, respectively, and the memory space was 39.0 MB and 20.6 MB respectively. In general, the execution time and memory space of our proposed method were within the acceptable range.

Table 7. The complexity of our method on different types of vulnerability.

Vulnerability	Pre-Training		Representation Learning		Classifier Training	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
CWE-119	216.5	278.0	301.1	2356.6	7.0	39.0
CWE-399			137.4	2341.9	4.2	20.6

5.6. Comparative Analysis

In order to examine the effectiveness of the proposed method in vulnerability detection, we performed a comparative experiment with state-of-the-art methods. We chose open-source static analysis tool Flawfinder [4] and commercial static analysis tool Checkmarx [5]. They represent traditional vulnerability detection methods based on static analysis. VUDDY [19] was chosen to represent the similarity-based method. VulDeePecker and our methods detect vulnerabilities based on learning vulnerable programming patterns. The difference is that VulDeePecker [25] has no representation learning stage, and it uses one dataset through the entire method. All the results in Table 8 are based on the same dataset. The results of Checkmarx and VulDeePecker are from the paper [25]. This is because we are not able to access the commercial tool Checkmarx and VulDeePecker is not an open-source tool.

Table 8. Comparative experimental results.

System	FPR (%)	FNR (%)	P (%)	R (%)	F1 (%)
Flawfinder [4]	46.3	69.0	23.7	40.5	29.9
Checkmarx [5]	43.1	41.1	39.6	58.9	47.3
VUDDY [19]	3.5	91.3	47.0	8.7	14.7
VulDeePecker [25]	2.9	18.0	91.7	82.0	86.6
Our method	1.5	9.6	95.7	90.4	93.0

It can be seen from Table 6 that our method outperforms the state-of-the-art methods. Specifically, traditional static analysis methods incur high FNR and FPR. The result of Checkmarx was relatively superior to Flawfinder. This is because the former applies efficient data flow analysis algorithms. VUDDY has a low FPR (3.5%), but has a poor performance on FNR (91.3%), P (47.0%), R (8.7%), and F1 (14.7%). This means that most vulnerabilities are not detected. This is because the similarity-based method is suitable for detecting vulnerabilities caused by code cloning, and is not adept at detecting general vulnerabilities. VulDeePecker and our method outperform other methods in most indicators.

This indicates that the vulnerable pattern-based method excels at detecting general vulnerabilities. Compared with VulDeePecker, our method improved by 1.4% in FPR, 8.4% in FNR, 4.0% in P, 7.6% in R, and 6.4% in F1. This indicates that compared with training classifier directly, the application of transfer learning and representation learning can effectively improve the effect of vulnerability detection.

The above experimental results show that our method has a fine granularity and representation ability by transforming the sample into the minimum intermediate representation. This facilitates audits and alleviates long-term dependency issues. By applying transfer learning, unlabeled data can be utilized to improve the effectiveness of vulnerability detection. By using the representation learning approach, our approach is able to learn the features of vulnerabilities from context. The comparative experiment results show that our method outperforms state-of-the-art methods. Compared with the traditional static analysis method, our method has a better performance. This is because traditional static analysis methods rely on expert-defined vulnerable patterns, which are laborious and error-prone due to the diversity of programming patterns. Compared with the similarity-based method, our method learns common vulnerability patterns from training samples and can be applied to a wide range of detection scenarios. Moreover, our method excels with other pattern-based methods because of the application of transfer learning and presentation learning. This proves our hypotheses.

6. Conclusions

This paper presents a novel solution to detect vulnerability in source code by learning vulnerable programming patterns automatically, which aimed to improve the effectiveness of vulnerability detection. The granularity of our proposed method is a minimum intermediate representation that extracts vulnerability relevant information based on dependency analysis. It covers not only intra-procedural but also inter-procedural vulnerabilities, and it alleviates the long-term dependency issue and provides precise vulnerability information. We transfer the common language features in an unlabelled dataset to the task of specific vulnerability detection. Representation learning is leveraged to abstract high-level features. It mitigates the impact of the OoV and the lack of vulnerability dataset issues. We implemented a prototype and performed systematic experiments to validate the effectiveness of our method. The experimental results show that our method has an obvious improvement over the state-of-the-art methods.

The proposed approach has several limitations which can be further investigated. First, the problem of long-term dependency is merely alleviated by eliminating the irrelevant code in the pre-processing stage. This is an inherent deficiency of the sequential structure. To improve this limitation, the graph embedding method can be applied. By transforming the source code into a graph structure, the element is connected directly to its context. Second, the method in this paper does not clean out the mislabeled samples in the dataset. Mislabeled samples will interfere with the learning of vulnerable programming patterns. Therefore, sample selection could be conducted in further research. Finally, our method falls into the category of static analysis. It means that our method cannot be applied to detect vulnerability in compiled software. This can be solved by transforming the compiled software into a common intermediate representation, such as the intermediate representation of Low Level Virtual Machine (LLVM).

Author Contributions: Conceptualization, X.L.; methodology, X.L.; software, X.L., L.W.; validation, X.L., L.W. and Y.X.; formal analysis, X.L., L.W.; investigation, X.L.; resources, Y.X., L.W. and Y.Y.; data curation, Y.X., Y.Y.; writing—original draft preparation, X.L.; writing—review and editing, X.L.; visualization, X.L., L.W.; supervision, Y.X., Y.Y.; project administration, Y.X.; funding acquisition, Y.X., Y.Y. and Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by “National Key R&D Program of China under Grant 2017YFB0802300”, “Major Scientific and Technological Special Project of Guizhou Province (20183001)”, “Foundation of Guizhou Provincial Key Laboratory of Public Big Data (No. 2018BDKFJJ021)”, and, “Foundation of Guizhou Provincial Key Laboratory of Public Big Data (No. 2017BDKFJJ015).”

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

References

1. WannaCry Ransomware Attack. Available online: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack (accessed on 29 December 2019).
2. National Vulnerability Database. Available online: <https://nvd.nist.gov> (accessed on 29 December 2019).
3. Brooks, T.N. Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In Proceedings of the Science and Information Conference, London, UK, 10–12 July 2018.
4. Flawfinder Software Official Website. Available online: <https://www.dwheeler.com/flawfinder/> (accessed on 29 December 2019).
5. CheckMarx Software Official Website. Available online: <https://www.checkmarx.com> (accessed on 29 December 2019).
6. Cadar, C.; Dunbar, D.; Engler, D.R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of the OSDI, San Diego, CA, USA, 8–12 December 2008.
7. Henzinger, T.A.; Jhala, R.; Majumdar, R.; Sutre, G. Software verification with BLAST. In Proceedings of the International SPIN Workshop on Model Checking of Software, Portland, OR, USA, 9–10 May 2003.
8. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [\[CrossRef\]](#)
9. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016.
10. Cyber Grand Challenge. Available online: <https://www.darpa.mil/program/cyber-grand-challenge> (accessed on 29 December 2019).
11. 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*, 56. [\[CrossRef\]](#)
12. Awad, Y.; Yashwant, M.; Charles, A.; Indrajit, R. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16), New Orleans, LA, USA, 9–11 March 2016.
13. Shin, Y.; Meneely, A.; Williams, L.; Osborne, J.A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.* **2010**, *37*, 772–787. [\[CrossRef\]](#)
14. Perl, H.; Dechand, S.; Smith, M.; Arp, D.; Yamaguchi, F.; Rieck, K.; Acar, Y. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015.
15. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* **2018**, *51*, 1–37. [\[CrossRef\]](#)
16. Wang, S.; Chollak, D.; Movshovitz-Attias, D.; Tan, L. Bugram: bug detection with n-gram language models. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, Singapore, 3–7 September 2016.
17. Grieco, G.; Grinblat, G.L.; Uzal, L.; Rawat, S.; Feist, J.; Mounier, L. Toward large-scale vulnerability discovery using machine learning. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, New Orleans, LA, USA, 9–11 March 2016.
18. Yamaguchi, F.; Maier, A.; Gascon, H.; Rieck, K. Automatic inference of search patterns for taint-style vulnerabilities. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015.
19. 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, 22–24 May 2017.
20. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016.
21. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017.

22. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y.; De Vel, O.; Montague, P. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Trans. Ind. Inform.* **2018**, *14*, 3289–3297. [[CrossRef](#)]
23. Pradel, M.; Sen, K. DeepBugs: A learning approach to name-based bug detection. In Proceedings of the ACM on Programming Languages (OOPSLA), Boston, MA, USA, 7–9 November 2018.
24. Bian, P.; Liang, B.; Zhang, Y.; Yang, C.; Shi, W.; Cai, Y. Detecting bugs by discovering expectations and their violations. *IEEE Trans. Softw. Eng.* **2018**, *45*, 984–1001. [[CrossRef](#)]
25. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 18–21 February 2018.
26. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z.; Wang, J. SySeVR: A framework for using deep learning to detect software vulnerabilities. *arXiv* **2018**, arXiv:1807.06756.
27. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Proceedings of the Advances in Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019.
28. Cummins, C.; Petoumenos, P.; Wang, Z.; Leather, H. Synthesizing benchmarks for predictive modeling. In Proceedings of the 2017 International Symposium on Code Generation and Optimization, Austin, TX, USA, 4–8 February 2017.
29. Allamanis, M.; Peng, H.; Sutton, C. A convolutional attention network for extreme summarization of source code. In Proceedings of the International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016.
30. Gu, X.; Zhang, H.; Kim, S. Deep code search. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018.
31. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
32. Brockschmidt, M.; Allamanis, M.; Gaunt, A.L.; Polozov, O. Generative code modeling with graphs. *arXiv* **2018**, arXiv:1805.08490.
33. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. In Proceedings of the Advances in Neural Information Processing Systems, Montréal, QC, Canada, 3–8 December 2018.
34. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014.
35. 2019 CWE Top 25 Most Dangerous Software Errors. Available online: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (accessed on 29 December 2019).
36. Pantel, P. Inducing ontological co-occurrence vectors. In Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics, Ann Arbor, MI, USA, 25–30 June 2005.
37. Gensim: Word2Vec Model. Available online: https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html (accessed on 29 December 2019).
38. SARD Manual. Available online: <https://samate.nist.gov/index.php/SARD.html> (accessed on 29 December 2019).
39. Common Vulnerabilities and Exposures. Available online: <https://cve.mitre.org> (accessed on 29 December 2019).

