


Article

Automated Software Vulnerability Detection Based on Hybrid Neural Network

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

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

² State Key Laboratory of Public Big Data, College of Computer Science and Technology, Guizhou University, Guiyang 550025, China

³ National Engineering Laboratory for Big Data Distribution and Exchange Technologies, Shanghai Data Exchange Corporation, Shanghai 200436, China; keven@chinadep.com

* Correspondence: li_xin@bupt.edu.cn (X.L.); ylchen3@gzu.edu.cn (Y.C.)

Abstract: Vulnerabilities threaten the security of information systems. It is crucial to detect and patch vulnerabilities before attacks happen. However, existing vulnerability detection methods suffer from long-term dependency, out of vocabulary, bias towards global features or local features, and coarse detection granularity. This paper proposes an automatic vulnerability detection framework in source code based on a hybrid neural network. First, the inputs are transformed into an intermediate representation with explicit structure information using lower level virtual machine intermediate representation (LLVM IR) and backward program slicing. After the transformation, the size of samples and the size of vocabulary are significantly reduced. A hybrid neural network model is then applied to extract high-level features of vulnerability, which learns features both from convolutional neural networks (CNNs) and recurrent neural networks (RNNs). The former is applied to learn local vulnerability features, such as buffer size. Furthermore, the latter is utilized to learn global features, such as data dependency. The extracted features are made up of concatenated outputs of CNN and RNN. Experiments are performed to validate our vulnerability detection method. The results show that our proposed method achieves excellent results with F1-scores of 98.6% and accuracy of 99.0% on the SARD dataset. It outperforms state-of-the-art methods.

Keywords: cyber security; vulnerability detection; program slice; static analysis



Citation: Li, X.; Wang, L.; Xin, Y.; Yang, Y.; Tang, Q.; Chen, Y. Automated Software Vulnerability Detection Based on Hybrid Neural Network. *Appl. Sci.* **2021**, *11*, 3201. <https://doi.org/10.3390/app11073201>

Academic Editor: David Megías

Received: 26 February 2021

Accepted: 30 March 2021

Published: 2 April 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The extensive application of information technology has significantly promoted the development of the economy and society. However, huge losses may be caused once an information system is attacked. Software vulnerabilities are the root cause of cyber-attacks. The number of software vulnerabilities published has grown rapidly in recent years. According to the 2020 annual report of US-CERT [1], national vulnerability database (NVD) recorded 17,447 vulnerabilities in 2020, which was the fourth consecutive year with a record number of vulnerabilities. Furthermore, some of them were utilized by attackers, causing enormous losses. For instance, more than 500,000 passwords were stolen and available for sale in April 2020 because of Zoom's vulnerabilities [2]. Therefore, it is crucial to find and patch vulnerabilities as early as possible.

Vulnerability detection is the work of finding code snippets that cause errors in certain situations in large chunks of code. Vulnerability detection remains a challenging and time-consuming task so far. Discovering potential vulnerabilities in the source code is the main branch of detection approaches. It can be applied in the whole life of software development. Traditional methods include taint analysis [3], symbolic execution [4], and theorem proving [5]. However, they suffer from low efficiency and high false positive rate. Moreover, intense labor of human experts is needed to make detection rules. Therefore,

automatic vulnerability detection is the trend of research. It leverages machine learning techniques to train the detector from vulnerability datasets. Metric- and pattern-based techniques are the two main branches of the intelligence approach. Inspired by the belief that complexity is the enemy of software security, metric-based detection approaches learn the connection between software engineering metrics and vulnerabilities. Complexity metrics [6], code churn metrics [7], developer activity metrics [8], and dependency metrics [9] are utilized. Metric-based approaches are lightweight to analyze a large-scale program. However, they suffer from high false positives and are mostly at source file granularity, requiring intense labor of human experts to locate the vulnerabilities. Pattern-based methods learn vulnerability patterns from source code using program language processing techniques. According to the learned patterns, they can be further divided into anomaly detection and vulnerable pattern learning. Anomaly detection learns legal programming patterns from mature applications at the training stage. Then, a code snippet will be reported as a potential vulnerability if it has an obvious low similarity with the learned legal pattern. Anomaly detection methods [10] avoid the dependency on labeled vulnerability datasets and can be applied to identify unknown vulnerabilities. However, their accuracy is relatively low, and a high false-positive rate will be made when used in a cross-project scenario. Vulnerable pattern learning formulates the vulnerable programming pattern from labeled vulnerability datasets. Compared with anomaly detection, vulnerable pattern learning requires smaller computations and performs better in accuracy and precision. However, it highly relies on labeled vulnerability datasets.

While the vulnerability detection method using machine learning technology has made significant progress in automation and accuracy, the following problems hinder its performance: (1) Long-term dependency between code elements. The dependencies between elements contain important information for vulnerability detection. However, semantically related elements may have a long distance from each other. For example, the assignment and reference of a variable may be interspersed with many irrelevant codes. The long-term dependency issue compromises the abstraction of vulnerability features. (2) The out of vocabulary (OoV) issue. Custom identifiers, such as function and variable names, result in a changing vocabulary. There are always new words out of vocabulary in a new project. (3) Coarse detection granularity. A detector with a fine granularity will promote the understanding and patching of vulnerabilities. However, most existing methods identify vulnerabilities at the level of file or function. Coarse granularity wastes the time and power of security experts to deal with the detection report. (4) Bias towards global features or local features. Methods in natural language processing are introduced into vulnerability detection. However, different neural network models have a different emphasis on feature extraction. For example, CNN tends to learn local features, while RNN is good at learning global features. Both global features or local features are significant for vulnerability detection. For example, both buffer size and data dependency play important roles in detecting buffer overflow vulnerability.

To this end, we propose an automated software vulnerability detection framework based on hybrid neural networks and the backward sliced intermediate representation method. This framework is designed to discover inter-procedural vulnerabilities hidden in program source code with fine granularity to pinpoint the vulnerability. The inputs in the form of source code are transformed into intermediate representations first. Specifically, each sample is compiled into LLVM IR, which is in the form of a static single assignment (SSA). LLVM IR has an explicit control dependency and data dependency. Then, a customized syntactic analysis is conducted to find all security sensitive program points in each input. These identified program points will serve as criteria to guide program slicing. Next, inter-procedural data dependency analysis and intra-procedural control dependency analysis are performed on each slice criterion. As a result, we abstract the vulnerability-related part of a sample. Subsequently, normalization is performed to replace the custom-defined identifiers with uniform ones and remove non-representative code and string variable definition. After the above steps, the inputs are transformed into intermediate represen-

tations with precise vulnerability information. This transformation significantly reduces the vulnerability context and the vocabulary size, relieving the long-term dependency and OoV issue.

Moreover, a hybrid neural network is proposed to learn high-level features of a vulnerability. A convolutional layer and a recurrent layer are applied. The convolutional layer is applied to learn local features, and the recurrent layer is utilized to learn global features.

We implemented the proposed framework and evaluated its performance on an authoritative vulnerability dataset. The experiment results show that our proposed intermediate representation facilitated the effect of the vulnerability detection method. The size of the samples and the size of vocabulary were significantly reduced after the transformation. The comparative analysis results show that our proposed method achieved excellent results with an F1-score of 98.6% and an accuracy of 99%. These results outperform state-of-the-art methods.

The remainder of this paper is structured as follows. In Section 2, we introduce some related noteworthy works. In Section 3, we present our proposed framework in detail. The experiments are discussed in Section 4. Finally, we conclude this paper and discuss future works in Section 5.

2. Related Work

This section reviews the related works from source code representation learning and intelligent vulnerability detection. We discuss source code presentation learning from sequence-based approaches, tree-based approaches, and graph-based approaches. Furthermore, intelligent vulnerability detection is reviewed from metric-based vulnerability detection and pattern-based vulnerability detection.

2.1. Source Code Representation Learning

Source codes must be transformed into a proper form to employ data-driven approaches for automatic vulnerability detection. This form should represent the source code's syntactic and semantic information and be suitable for further analysis in machine learning algorithms. There are three main representation learning methods for source code: sequence-based approaches, tree-based approaches, and graph-based approaches [11].

Sequence-based approaches treat code fragments as sequences, which is similar to natural language processing in a way. The elements that compose a sequence can be character, token, or application programming interface (API). Local embedding methods are proposed at first. Scandariato et al. [12] apply the “bag-of-words” model to encode textual tokens of each Java file to find vulnerabilities. Their implementation demonstrates the possibility of predicting software vulnerability using machine learning approaches. Pang et al. [13] further improved the detection effect by leveraging the N-gram model to encode source code. Although the local representation methods demonstrate their ability to represent source code in the vulnerability detection task, they usually suffer from high dimension and poor abstraction of long-term contextual information. Distributed representation methods are proposed to overcome these challenges. Li et al. [14,15] utilized word2vec to represent code fragments. Tokens are embedded using their context information. Towards the lack of vulnerability samples, a fine-tuned model was applied [16]. This model learns the embedding of tokens using a pre-training stage on an expanded source code corpus. The pre-training method's application reduces training time and improves performance using the transferred pre-known information. Sequence-based approaches have the advantage of a lightweight computation. However, the existing sequence-based methods are prone to the long-term dependency problem.

Tree-based approaches learn the representation from an abstract syntax tree (AST). AST is a kind of tree to abstract a source code with an explicit syntactic structure. On the one hand, AST serves as an intermediate representation between a source code and sequence to facilitate the representation using syntactic knowledge. Lin [17,18] proposed a function level representation method using AST. Functions in the form of AST are serialized

to sequence using depth-first traversal. Bilgin [19] used full binary AST for simplification and efficiency. Breadth-first traversal is leveraged to serialize binary AST into a sequence. Code2vec [20] and Code2seq [21] decompose AST to a set of paths using a random walk. Then the skip-gram model is applied to generate embeddings.

On the other hand, tree-based neural networks are applied directly to AST. Mou [22] leverages a custom convolutional neural network on a full binary tree of code snippets. To relieve the gradient limitation, an AST neural network (NN) [23] splits large ASTs into a set of small statement trees. Then, the tree-based RNN is performed. Overall, tree-based approaches improve the representation of source code by adding syntactic knowledge. Furthermore, arbitrary code fragments can be parsed to AST, which broadens the usage scenarios of tree-based approaches. However, tree-based methods dramatically increase code fragment complexity. Moreover, they are prone to the long-term dependency problem and the gradient vanishing problem.

Graph-based approaches parse programs in the form of source code into graph representations with structure knowledge. Then graph-based embedding methods are applied to the graph representations. Tufano et al. [24] learned similarities between different code snippets using a control flow graph (CFG). Allamanis et al. [25] used the combination of AST and data flow graph (DFG) as a graph representation before embedding. Furthermore, a program dependency graph (PDG) [26], code property graph (CPG) [27], and context flow graph (XFG) [28] were also applied to the representation learning of programs.

Graph-based approaches relieve the long-term dependency issue and improve the performance of detectors. However, they need intensive computation, which requires a large number of hardware resources. Moreover, most graph-based methods rely on the compiled intermediate representation limiting their application scenarios.

2.2. Intelligent Vulnerability Detection

The development of artificial intelligence technology brings new methods to alleviate the bottlenecks of traditional vulnerability detection. Metric-based and pattern-based approaches are the two main branches. Inspired by bug prediction, software engineering metrics are used as features to detect vulnerabilities. At first, the successful application of complexity metrics [6,29] reveals the capacity of metric-based approaches to discover vulnerabilities. Subsequently, more metrics such as token frequency metrics [30], execution complexity metrics [31], developer activities metrics [7], and dependency metrics [9] are utilized to improve the performance of a detector. Recently, Du et al. [32] used vulnerability metrics in an additional step. This method achieves excellent results in cross-project situation. Metric-based vulnerability detection is a lightweight method to find vulnerabilities in source code at file-level granularity. However, their recall is low, and file-level granularity is too coarse to help the developer locate and patch the reported vulnerabilities.

Pattern-based approaches are proposed to improve efficiency and reduce reliance on human workers. On the one hand, normative patterns derived from mature software are utilized as rules to find vulnerabilities. Violations from these norms are considered potential vulnerabilities. Fabian [33,34] derived anomalous patterns from the variable initializations with data or control dependency on security-sensitive functions. Some key statements or API usages are collected, such as abnormal API usage patterns, imports, and function calls. Bian et al. [35] utilized syntactic information to derive anomalous patterns. They transform sliced source codes into AST. Then, a hash algorithm is used to obtain the vector representation. Anomaly detection has the advantages of finding unknown vulnerabilities, and it reduces the dependence on labeled vulnerability datasets. However, anomaly detection is task/project-specific and incurs a low recall.

On the other hand, vulnerable programming patterns derived from vulnerable programs are also applied to detect potential vulnerabilities. Dam et al. [36] proposed a file-level vulnerability detection method. They use long short-term memory (LSTM) to learn the embedding of a serialized AST. Local features and global features are leveraged to train a simple classifier. Wang et al. [37] parsed files into AST nodes and leveraged a

deep belief network (DBN) to learn semantic features from token vectors. Vulnerability detections with a granularity of file-level has the advantage of easily acquired datasets. However, its detection granularity is too coarse for location and understanding of reported vulnerabilities.

Liu and Lin [38,39] utilized transfer learning to minimize the divergence between the source domain and the target domain. Their approach performs well in cross-project vulnerability detection with a function-level granularity. VulSniper [27] leverages CFG and AST as a graph representation and uses soft attention to learn high-level features of vulnerabilities. Devign [26] proposed a function-level detection method using a graph representation that combines AST, dependency, and natural code sequence information. Function-level vulnerability detections provide precise vulnerability information compared with file-level ones. However, they fail to capture inter-procedural semantic details due to their granularity.

Inter-procedural statement-level vulnerability detection methods are proposed with a fine granularity. SySeVr [15] and VulDeePecker [14,40] utilize inter-procedural dependency analysis to generate sliced source code, and word2vec is used as the embedding method. VulDeeLocator [41] transforms source code into the LLVM IR form, representing syntax and semantic information in the form of an SSA. Then, the attention mechanism and LSTM neural networks are leveraged to learn high-level features. Compared with file-level and function-level methods, those with inter-procedural statement-level granularity can abstract broad scope vulnerability types and provide precise information for the location and understanding of the reported vulnerabilities. However, they are relatively complex due to the inter-procedural dependency analysis.

3. Methods

In this section, we describe in detail how our proposed method automatically learns high-level features and discovers potential vulnerabilities in source code.

3.1. Problem Formulation

A software vulnerability is a fault or mistake created during development. Malicious users may exploit it to perform functions different from design logic. In this paper, we focus on detecting vulnerabilities in source code. Most vulnerabilities are rooted in critical operations. A critical operation can be a function call, an assignment, or a control statement. Therefore, from the source code perspective, we describe a vulnerability as follows:

Definition 1. *In source code, a vulnerability is a security-sensitive operation and the statements that have control or data dependency on it. These relative statements can be inter-procedural or intra-procedural.*

Vulnerability detection is essentially a binary classification problem. Our goal is to design a detector that can classify a program into a correct label based on the knowledge learned from labeled datasets. It can be described as follows:

Let a sample be defined as $((c_i, y_i) | c_i \in C, y_i \in Y), i \in \{1, 2, \dots, n\}$, where C represents the set of programs and $Y = \{0, 1\}^n$ denotes the label of corresponding programs with 0 for clean and 1 for vulnerable. n is the number of programs in datasets. Let R denote the intermediate representations that are fed to the machine learning model. The function $g : C \rightarrow R$ parses programs in raw source data into intermediate representations. The goal of our method is to learn a mapping from R to Y , $f : R \rightarrow Y$ to predict whether a program is vulnerable or not.

3.2. Overview

Our study aimed to detect vulnerabilities in source code with a fine granularity. Therefore, we chose the inter-procedural statement-level granularity. As shown in Figure 1, our detection framework goes through the training phase and detecting phase. At the

training phase, programs in the form of source code are parsed into LLVM IR, which is in the form of an SSA. SSA provides explicit use-define chain and control dependency in the context. Then, an inter-procedural static analysis on parsed intermediate representations is conducted. Our framework performs syntactic analysis to find all security-sensitive operations. Some security-sensitive operations may belong to one vulnerability, so vulnerability description information is utilized to find these key operations as slice criteria. Next, backward program slicing is performed using these slice criteria. The slicing results are treated as the input of a hybrid neural network after normalization and serialization. In the hybrid neural network, the recurrent layer is used to learn the long-term information, while the convolutional layer is applied to learn the local details of a program. The outputs of the recurrent layer and the convolutional layer are concatenated as the input of a dense layer, which generates the detection result. At the detection phase, the target programs will be parsed into intermediate representations and classified by the trained model.

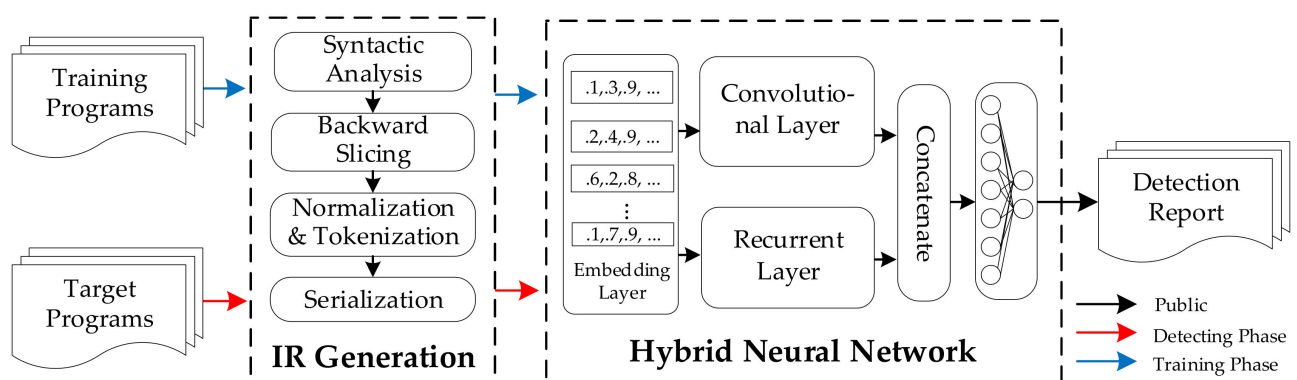


Figure 1. Overview of the proposed vulnerability detection approach. The red arrow represents the data flow in the training phase. The blue arrow represents the data flow in the detecting phase. The black arrow represents the data flow in both the training and detecting phases.

3.3. Intermediate Representation Generation

Programs are transformed into intermediate representations that have explicit data dependency and control dependency first. Irrelevant information is removed to reduce the interference of noise. To do so, LLVM IR and backward program slicing is applied.

3.3.1. Parsing Source Code

A program language has strict syntax rules and dependency relations. This characteristic is crucial to the representation of vulnerabilities. However, dependency relations in a program source code may be implicit. For example, a variable may be defined and used in more than one place. Furthermore, these options all share one name. This issue may incur confusion while using the automated detection method to understand the program.

In our method, LLVM IR is utilized as an initial intermediate representation to obtain explicit dependency relations, which has already been used by compilers for optimization. LLVM IR is given in static single assignment (SSA) form, ensuring every variable is defined only once. Each use of a variable is assigned a new identifier, and all these identifiers are connected to one common register. This form makes it easy for machine learning models to understand the dependency in a program. As shown in Figure 2, a simple function written in the C program language in Figure 2a is transformed into LLVM IR form in Figure 2b. In LLVM IR, custom-defined variables are renamed to the form of a specific token plus a number. For example, local variables are renamed by `"%ID"` and global variables are renamed by `"@ID"`. This naming method ensures a fixed vocabulary, relieving the out of vocabulary issue. Moreover, control flow is specified by `"br"` and `"label"` in LLVM IR. These characters facilitate machine learning methods to understand a program.

```

1  #include <stdio.h>
2
3  int main() {
4
5      int x = 1;
6      int y;
7
8      if (x > 1) {
9          y = 2;
10     }
11
12     else{
13         y = 3;
14     }
15
16     return 0;
17 }
18
19

```

(a) Source code

```

1  define i32 @main() #0 {
2      %1 = alloca i32, align 4
3      %2 = alloca i32, align 4
4      %3 = alloca i32, align 4
5      store i32 0, i32* %1, align 4
6      store i32 1, i32* %2, align 4
7      %4 = load i32, i32* %2, align 4
8      %5 = icmp sgt i32 %4, 1
9      br i1 %5, label %6, label %7
10
11     ; <label>:6:                                ; preds = %0
12     store i32 2, i32* %3, align 4
13     br label %8
14
15     ; <label>:7:                                ; preds = %0
16     store i32 3, i32* %3, align 4
17     br label %8
18
19     ; <label>:8:                                ; preds = %7, %6
20     ret i32 0
21 }

```

(b) LLVM IR

Figure 2. A code snippet and its lower level virtual machine intermediate representation (LLVM IR): (a) A program sample written in C language; (b) the corresponding LLVM IR form of the program listed in Figure 2a.

3.3.2. Finding Slice Criterion

Our proposed framework chooses inter-procedural statements as detection granularity. Hence, further static analysis is needed. Furthermore, vulnerability descriptions are used to facilitate the generation of the intermediate representation. As discussed in Section 3.1, the key part of a vulnerability is the set of statements that perform security-sensitive operations. Therefore, these statements are found as criteria to guide program slicing. Firstly, the label and line number of the statements that perform security-sensitive operations are extracted from vulnerability descriptions. Traditional static analysis methods already summarize security-sensitive operations. Then, syntax analysis is performed on every found statement to pinpoint the call of security-sensitive functions or memory operations, which are extracted as slice criteria. The whole algorithm is described in Algorithm 1.

Algorithm 1. Extracting security-sensitive operations from a program.

Input: A program $p = \{s_1, s_2, \dots, s_m\}$, vulnerable syntax rules $R = \{r_1, r_2, \dots, r_k\}$, Vulnerability manifest $M = \{m_1, m_2, \dots, m_l\}$

Output: A set C of slice criterions

```

1:  $C \leftarrow \emptyset$ ;
2: for each vulnerability description  $m_i \in M$  do:
3:    $line \leftarrow$  Get line from  $m_i$ 
4:    $label \leftarrow$  Get label from  $m_i$ 
5:   for each rule  $r_i \in R$  do:
6:     if  $s_{line}$  match  $r_i$ :
7:        $c \leftarrow$  extract slice criterion from  $s_i$ ;
8:        $C \leftarrow C \cup \{(c, line, label)\}$ ;
9:     end if;
10:  end for;
11: end for;
12: return  $C$ ;

```

3.3.3. Program Slicing and Serialization

In this paper, the token sequence of sliced LLVM IR is leveraged as the final intermediate representation. A backward program slicing is performed to remove instructions of a program that do not influence the slice criterion. The generation of program slice relies on data dependency and control dependency, which define how other instructions affect some program points of interest.

Definition 2. Data dependence. A statement j is data dependent on statement i (written $i \xrightarrow{dd} j$) if there exists a variable x such that: (1) $x \in \text{DEF}(i) \cap \text{REF}(j)$, (2) there exists a non-trivial path p from i to j such that for every statement $k \in p - \{i, j\}$, $x \notin \text{DEF}(k)$. $\text{DEF}(i)$ denotes the set of variables defined at statement i , and $\text{REF}(j)$ denotes the set of variables referenced at statement j .

Definition 3. Control dependence. A statement j is control dependent on a statement i (written $i \xrightarrow{cd} j$) (1) if there exists a path p from i to j such that j post-dominates every statement $k \in p - \{i, j\}$, (2) i is not post-dominated by j .

Data dependence information identifies the statements that modify the value of a variable in a program point. Control dependence information identifies the conditionals that may affect the execution of a statement.

Definition 4. Backward program slice. Let C be the slice criteria of a program. Backward program slice with respect to a slice criterion $c \in C$ is defined as a set $S_c = \{s \mid s \xrightarrow{dd} c \text{ or } s \xrightarrow{cd} c\}$.

Based on the definition of control dependence and data dependence, all the instructions that can affect a given criterion are found. Then, preprocessing that removes non-representative code and normalizes custom-defined identifiers from the backward program slice of a program is conducted. After this step, the length of a sample is reduced, relieving the out of vocabulary issue. Finally, tokenization is performed on the natural sequential order of the preprocessed backward program slice. The results are fed to a hybrid neural network to learn the vulnerable pattern automatically.

3.4. Hybrid Neural Network

As shown in Figure 3, a hybrid neural network is proposed to learn the vulnerable pattern automatically. RNN has proved its ability to process sequential order data tasks. However, compared with natural language processing and program summary, local details play a crucial role in vulnerability detection. For example, a vulnerable program may have little difference from its patched version. The cause of a vulnerability could be just a few misused variables or control statements. CNN has the advantage of learning local features. Therefore, our proposed hybrid neural network utilized a convolutional layer to learn local features and a recurrent layer to learn global features.

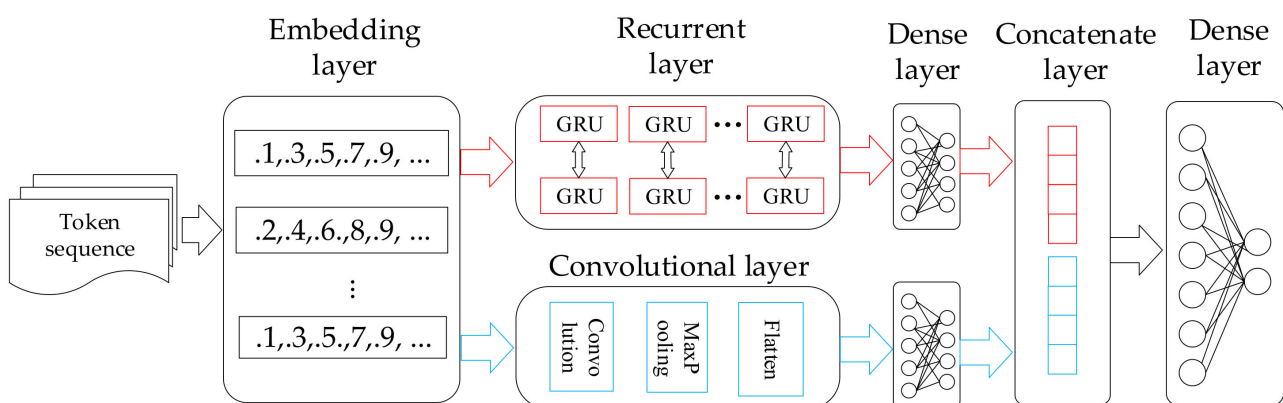


Figure 3. Overview of the proposed hybrid neural network. The input token sequence is converted into vector representation in the embedding layer. Then it is input into the recurrent layer and the convolutional layer, respectively. The red arrows represent the process of the recurrent layer, and the blue arrows represent the process of the convolutional layer. The outputs of these two layers are concatenated and feed into a dense layer to predict whether an input sample is vulnerable or not.

As shown in Figure 3, our proposed neural network consists of six layers. Specifically, the network takes a token sequence $\langle t_1, t_2, \dots, t_n \rangle$ generated from Section 3.3 as input, where n is the length of a sequence. Each token $t_i \in R^V$ in the input is a vector in the form of one-hot, where V is the vocabulary size. The embedding layer maps each token into a fixed-length continuous vector, which is done by maintaining a token embedding matrix $E \in R^{k \times |V|}$, where k is the size of a token vector, and $|V|$ is the size of the vocabulary V . The embedding matrix that acts as a look-up table is randomly initialized and then modified during training. The vector representation of a token t_i is generated by

$$v_i = t_i \times E \quad (1)$$

where i is the index of a token in a token sequence. Then the input can be represented as

$$x_j = v_1 \oplus v_2 \oplus \dots \oplus v_n \quad (2)$$

where j is the order of a sample in the dataset, and \oplus denotes the concatenate operation. The vector representation of input is sent to the convolutional layer and the recurrent layer, respectively.

In the convolutional layer, let $w \in R^{h \times k}$ be a filter, where h is the filter window. A new feature c_i is produced by

$$c_i = f(w \cdot x_{i:i+h-1} + b) \quad (3)$$

Here, $x_{i:i+h-1}$ denotes the token vectors that are involved in one convolution, and $b \in R$ is the bias. f denotes a non-linear function. This filter is applied to each possible window in the input token sequence, and a feature map is produced by

$$C = [c_1, c_2, \dots, c_{n-h+1}]. \quad (4)$$

A max-pooling operation with a pooling size d is performed over the feature map. After this special filter, all d features are converted into one feature by maximizing value on every dimension. Last, we apply a flattening operation to convert the feature map into a vector $\hat{C} \in R^{(n-h+1)k/d}$. In the recurrent layer, the Bi-directional Gated Recurrent Unit (Bi-GRU) network is utilized to capture the long-term dependencies for the input sequence. Every vector representation v_t is input into a GRU unit. As described in [42], the basic recurrence of the propagation model is

$$\begin{aligned} r_t &= \partial(v_t W_{vt} + h_{t-1} W_{hr} + b_r) \\ z_t &= \partial(v_t W_{vz} + h_{t-1} W_{hz} + b_z) \\ \tilde{h}_t &= \tanh(v_t W_{vh} + r_t \odot h_{t-1} W_{hz} + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \end{aligned} \quad (5)$$

where r_t and z_t represent reset and update gates, respectively. \tilde{h}_t denotes the candidate hidden state, and h_t is the output of the recurrent GRU unit. Furthermore, the symbol ∂ denotes the sigmoid function, and \tanh represents the hyperbolic tangent function. The symbol \odot signifies element-wise multiplication. GRU captures long-term dependencies using update gate and reset gate. Furthermore, Bi-GRU implementation, which consists of a forward and a backward GRU, is applied to capture both preceding and subsequent contextual information. The recurrent layer output $H \in R^{2u}$ can be calculated using $H = H_f \oplus H_b$, where u is the number of the GRU unit, and H_f, H_b represent the output of forward and backward GRU, respectively.

In the last three layers, the outputs of the recurrent layer and the convolutional layer are converted into fixed-length vectors using a fully connected neural network, respectively. After that, these two vectors are concatenated and finally input into a dense layer to output vulnerable and clean probabilities.

3.5. Vulnerability Detection

In the detection phase, target programs are transformed into the intermediate representation in the same way as that of the training phase. The model learned in the training phase is performed as a classifier, generating predictions according to the transformed input.

The proposed syntax analysis method is applied directly to find all security-sensitive program points for the new programs with no vulnerability description. Each program point serves as a criterion to generate the corresponding program slice. Finally, program slices are classified by the trained model.

Our method traces back a detected vulnerability to the respective piece of source code to help developers understand and fix vulnerabilities. As described in Section 3.2, the key part of a vulnerability is the security sensitive program point from the source code perspective. After the detection phase, vulnerable samples in sliced LLVM IR are detected. Security sensitive program points are extracted from the detected samples. Then these program points act as criteria to perform traditional static analysis at the source code level, such as data flow analysis. Together with the static analysis results, the extracted security sensitive program points make up the final report.

4. Experiments

4.1. Research Questions

We conduct experiments toward answering the following research questions:

RQ1: Can our proposed intermediate representation facilitate the feature extraction of vulnerabilities from source code?

RQ2: How well does our approach perform in learning vulnerable program patterns?

RQ3: How effective and precise is our proposed method compared with the state-of-the-art methods?

4.2. Evaluation Metrics

To measure our proposed method's performance, a confuse matrix was utilized to record the correct and incorrect predictions. Let True Positive (TP) denotes the number of vulnerable samples that are correctly classified. False Positive (FP) denotes the number of clean samples that are classified as vulnerable. True Negative (TN) represents the clean samples that are correctly classified as clean. False Negative (FN) represents the vulnerable samples that are falsely classified as clean. False Positive Rate (FPR) = $FP / (TN + FP)$, denotes the proportion of clean samples that are falsely classified as vulnerable in all clean samples. False Negative Rate (FNR) = $FN / (FN + TP)$ denotes the ratio of falsely classified vulnerable samples on all vulnerable samples. Accuracy (Acc) = $(TP + TN) / (TP + TN + FP + FN)$, represents the ability to classify vulnerable and clean samples correctly. 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 detection method to discover vulnerabilities as much as possible. F1 – Score (F1) = $2 \cdot P \cdot R / (P + R)$ is an indicator that takes both Precision and Recall into consideration.

4.3. Experimental Setup

We used the Software Assurance Reference Dataset (SARD) [43] as the data source because it has been widely used as a benchmark to test vulnerability detection methods. SARD is a project maintained by the National Institute of Standards and Technology (NIST). Their samples include synthetic, production, and academic vulnerabilities. A clear vulnerability description was accessed in SARD, which is crucial for our method to perform preprocessing before machine learning. Specifically, 12,301 programs in the C language were chosen, including 11 types of vulnerabilities: CWE-20, CWE-78, CWE-119, CWE-121, CWE-122, CWE-124, CWE-126, CWE-127, CWE-134, CWE-189, and CWE-399. Moreover, the vulnerabilities may be inter-procedural or intra-procedural. We set train data:test data = 7:3, and 5-fold cross-validation was applied to choose super parameters.

Clang (version 6.0.0-1ubuntu2) [44] was used to parse programs into LLVM IR form. DG (version 0e0fc8f9) [45] was used to conduct backward program slicing on LLVM IR. Because DG is still under development, it takes all calls of function with the same name as slice criteria in the current version. The other function calls are also designed as candidate vulnerabilities in SARD. Thus, we removed all calls to the same function except the selected security-sensitive program point before slicing. After normalization and tokenization, the vocabulary size on the data source was reduced to 537. After compiling and static analysis, we extracted 32,860 code snippets, including 12,301 vulnerable ones and 20,559 clean ones. A hybrid neural network was implemented by Keras on Kaggle with NVIDIA Tesla P100 GPU. The tuned parameters of the hybrid neural network are shown in Table 1.

Table 1. Tuned parameters of hybrid neural network.

Parameter	Description
Embedding layer	The size of vocabulary and token vector are 538 and 100, respectively. The embedding matrix is randomly initialized. The max sequence length is 1000.
Convolutional layer	The number of convolutional filters is 256 with size 3. The window of the max-pooling filter is 4.
Recurrent layer	The units of both forward and backward GRU net are 256. Dropout and recurrent_dropout are set to 0.2 and 0.1, respectively.
Dense layer	The first two dense layers with 256 units are connected to the convolutional layer and the recurrent layer. The last dense layer generates prediction with two units.
Batch_size	The number of samples that are propagated through the network is 128.
Loss function	The function to evaluate the difference between the trained model and datasets is binary_crossentropy.
Optimizer	The algorithm to optimize the neural network is Adam.
Monitor	The metric to be monitored for early stop is F1-score and patience is 10.

4.4. Experiments for Answering RQ1

In order to evaluate the effect of our proposed intermediate representation, experiments were conducted on five datasets. All five datasets were generated from the same dataset. Specifically, “raw data” denotes the raw dataset described in Section 4.3, which consists of programs with vulnerability and corresponding patches in their bodies. Moreover, a sample could include multiple source code files. “LLVM IR” denotes the intermediate codes that were transformed from “raw data”. “Source code based” consists of program slices in the form of source code. It was transformed from “raw data” using our slice criterion-finding method. “VulDeeLocator” is a dataset that was generated by VulDeeLocator [42], which is also an LLVM IR and program slicing based method. Table 2 summarizes these five datasets from the number of samples, the average length of a sample, the vocabulary size, and the F1-score. The indicator F1-score obtained by applying the same neural network on these datasets represents the ability of intermediate representation to promote vulnerability detection.

Table 2. Results for RQ1. Evaluating the performance of different intermediate representations.

Data	Sample Number	Average Length	Vocabulary Size	F1-Score
Raw data	12,301	185	60,918	-
LLVM IR	12,301	513	43,179	-
Source code based	32,860	144	691	88.9
VulDeeLocator [42]	140,631	34	2665	97.4
Our method	32,860	51	537	98.6

As shown in Table 2, “raw data” had 12,301 samples in total. The average length of a sample was 185, and the vocabulary size was 60,918. After compiling, the average length increased to 513 in “LLVM IR”, because LLVM IR was in the form of an SSA, and non-representative codes were introduced. Both “raw data” and “LLVM IR” had no F1-score because vulnerability and patch were mixed in some samples. As for “source code based”, 32,860 samples were extracted from “raw data” using our proposed slice criterion finding method. The average length of a sample and the vocabulary size were reduced to 144 and 691, respectively, after normalization. “VulDeeLocator” had 140,631 samples, which is approximately four times that of “our method”, because “VulDeeLocator” travels the entire program to find all security-sensitive points whose bidirectional program slice includes a vulnerable line given in the vulnerability description file. This method will incur a situation in which multiple samples may belong to one vulnerability. The generated slices are inaccurate and fragmented. Moreover, “VulDeeLocator” normalizes custom-defined function names, reducing the vocabulary size from 43,197 to 2665. “Our method” was based on LLVM code and extracted 32,860 samples. Each sample corresponded to a vulnerability or a patch in the program. The average length of a sample was 51, 1/10 of that of “LLVM IR”. In the normalization of “our method”, all the custom-defined identifiers, such as structure, union, and databuf, were replaced with unified names, reducing the vocabulary size to 537. Our method achieved an F1-score of 98.6%, which outperformed other methods, proving our proposed intermediate representation’s effectiveness.

4.5. Experiments for Answering RQ2

Experiments were performed to compare our method with three embedding methods to verify our method’s embedding effect. Specifically, “vocabulary” represents the local embedding method, which utilizes the vocabulary model directly to encode the inputs. A 4-layer fully connected neural network was applied to generate the detection results. The last three methods are distributed embedding methods. They use our hybrid neural network as a representation learning model. “Random” denotes the embeddings of inputs randomly initialized. “Static” denotes the embeddings of inputs generated by word2vec in the pre-training stage. Furthermore, the embedding matrices remain static in the training stage. “None-static” denotes the embeddings of inputs generated by word2vec in the pre-training stage. The embedding matrices were modified in the training stage. The results are shown in Table 3.

Table 3. Comparison of 4 different embedding approaches.

Method	FPR (%)	FNR (%)	Acc (%)	P (%)	R (%)	F1 (%)
Vocabulary	12.9	80.3	84.1	79.7	73.8	76.6
Random	1.0	1.0	99.0	98.3	99.0	98.6
Static	1.4	3.6	87.8	97.6	96.4	97.0
None-static	0.9	3.0	98.3	98.4	97.0	97.7

As shown in Table 3, the distributed embedding method had a significant improvement in vulnerability detection effectiveness compared with the local embedding method. Using the same neural network, “random” achieved the best F1-score, recall, accuracy, and false negative rate. “None-static” achieved the best performance on false positive rate and precision. In general, “random” achieved the best embedding effect on the SARD dataset.

To evaluate our proposed hybrid neural network’s ability, experiments were performed to compare our method with three neural networks, which have been proven effective in processing sequence data. All experiments were conducted on the same dataset processed by our method discussed in Section 3.3. Neural networks for comparison were Text-CNN, Bi-GRU, and GRU + Attention. They all use the “random” embedding method. The results are shown in Table 4. Six indicators were utilized to measure the effectiveness of a network. Specifically, “Acc”, “P”, “R”, and “F1” are described in Section 4.2. “TTime” denotes the training time that neural networks cost in an epoch. “DTime” denotes the

detecting time of the trained model on 9858 samples. Target samples and training samples are different vulnerabilities and are both from the SARD dataset.

Table 4. Results for RQ2. Comparison of 4 different neural networks.

Model	Acc (%)	P (%)	R (%)	F1 (%)	TTime (s)	DTime (s)
Text-CNN	92.6	95.6	89.8	92.6	11.0	1.5
Bi-GRU	98.2	96.5	98.8	97.6	203.6	109.8
GRU + Attention	93.8	90.5	93.2	91.8	831.2	180.9
Our model	99.0	98.3	99.0	98.6	1086.7	122.1

As shown in Table 4, “Text-CNN” had a minimum training time of 11.0 s/epoch and a detecting time of 1.5 s. However, it had the lowest accuracy and recall. “Bi-GRU” made a significant improvement compared with “Text-CNN”. “GRU + Attention” had the worst performance on precise and F1-score. Our proposed hybrid neural network achieved the best results on accuracy, precision, recall, and F1-score, which were 99.0%, 98.3%, 99.0%, and 98.6% respectively. Although our model took 1086.7 s/epoch to train the network, the improvement was clear. Vulnerability detection aims to discover potential vulnerabilities as much as possible. Therefore, the training time for our method is acceptable.

4.6. Experiments for Answering RQ3

In order to answer RQ3, we compared our proposed method with five state-of-the-art vulnerability detection methods. “Flawfinder” [46] is a famous open-source vulnerability detector using static analysis. “Checkmarx” [3] is a commercial tool based on dependency analysis. “Flawfinder” and “Checkmarx” represent traditional vulnerability detection methods. Both of them rely on the vulnerable programming patterns defined by experts. “SySeVr” [15] is a source code-based detection method. Program slice is applied to generate candidates, and a Bi-GRU network is utilized to learn vulnerable programming patterns automatically. “VulDeeLocator” [41] is an automatic method that uses sliced LLVM IR as candidates and utilizes Bi-GRU and attention mechanisms to learn vulnerable programming patterns. Both “SySeVr” and “VulDeeLocator” are sequence-based methods. “Devign” [26] is a graph-based detection method. The samples were transformed into a graph representation, and then GGRN was applied to learn vulnerable programming patterns automatically. The results are listed in Table 5.

Table 5. Results for RQ3. Comparative analysis.

Method	FPR (%)	FNR (%)	Acc (%)	P (%)	R (%)	F1 (%)
Flawfinder [46]	12.9	80.3	62.1	47.3	19.7	27.8
Checkmarx [3]	46.5	51.8	51.6	37.7	48.2	42.3
SySeVr [15]	10.6	11.5	89.1	83.1	88.5	85.7
VulDeeLocator [41]	1.3	4.1	97.7	97.8	95.9	96.8
Devign [26]	8.3	13.3	89.8	86.0	86.7	86.4
Our method	1.0	1.0	99.0	98.3	99.0	98.6

As shown in Table 5, compared with vulnerable pattern learning methods, the traditional approaches “Flawfinder” and “Checkmarx” had low detection performance. Their poor performance was because traditional methods detect vulnerabilities based on expert-defined rules, which cannot fit all situations. “SySeVr” achieved a 37.5%, 45.4%, 40.3%, and 43.4% improvement over “Checkmarx” in accuracy, precision, recall, and F1-score, respectively. This improvement demonstrates the ability of the sequence-based machine learning method. “VulDeeLocator” achieved an 8.6%, 14.7%, 7.4%, and 11.1% improvement over “SySeVr” in accuracy, precise, recall, and F1-score, respectively. This improvement demonstrates the potential of LLVM IR for vulnerability detection. “Devign” achieved a slight improvement over “SySeVr”, which was because the application of graph representation learning highlighted the structure information. However, “Devign” takes a statement

as the base element of a graph. Each statement is obtained by calculating the average of all inside tokens. This operation will lose the local details, which is crucial for vulnerability detection. Our method achieved 99.0%, 98.3%, 99.0%, and 98.6% in accuracy, precision, recall, and F1-score, respectively, reaching a new state-of-the-art on the SARD dataset. This is attributed to the proposed intermediate representation and the hybrid neural network, which takes both long-term dependencies and local details.

5. Conclusions

In this paper, a novel approach that detects source code vulnerabilities automatically is proposed. The programs are transformed into intermediate representations first. LLVM IR and backward program slicing are utilized. The transformed intermediate representation not only eliminates irrelevant information but also represents the vulnerabilities with explicit dependency relations. Then, a hybrid neural network is proposed to learn both local and long-term features of a vulnerability. We have achieved a prototype. The experiment results show that our approach outperforms state-of-the-art methods.

The proposed approach has several limitations, which can be further investigated. Firstly, our method is applied to detect vulnerabilities in source code written in C language at present. In theory, our approach can be applied to other programming languages as well. Therefore, applying our methods to other languages is one of the interesting future works. Secondly, our approach is only conducted on the SARD dataset due to the lack of labeled vulnerability datasets and falls into in-project vulnerability detection. The lack of labeled datasets is an open problem restricting the development of automated vulnerability detection technology. Existing vulnerability datasets suffer from the wrong labels and coarse-grained vulnerability descriptions. For example, the VDISC dataset is generated by the traditional static detection method. However, the conventional static detection method itself has the problem of a high false positive rate. The NVD dataset only provides the difference between the vulnerability sample and the patch. It is difficult to locate the security sensitive program point. Therefore, one of the interesting future works is proposing a method to generate labels on real world software and transfer the learned vulnerable patterns to detect vulnerabilities in a new project. Transfer learning and graph-based representation learning method should be applied in future work to solve the cross-project task. Thirdly, the inputs processed by our method should be able to be compiled. This means further effort is needed for compiling when only a code snippet is accessible.

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

Funding: This research was funded by the “Major Scientific and Technological Special Project of Guizhou Province (20183001)”, and the “Foundation of Guizhou Provincial Key Laboratory of Public Big Data (2017BDKFJJ015, 2018BDKFJJ020, 2018BDKFJJ021)”.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data available in a publicly accessible repository that does not issue DOIs. Publicly available datasets were analyzed in this study. This data can be found here: <https://samate.nist.gov/index.php/SARD.html> (accessed on 2 April 2021).

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

References

- CVSS Severity Distribution Over Time. Available online: <https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time> (accessed on 19 January 2021).
- Over 500,000 Zoom Accounts Sold on Hacker Forums the Dark Web. Available online: <https://www.bleepingcomputer.com/news/security/over-500-000-zoom-accounts-sold-on-hacker-forums-the-dark-web/> (accessed on 19 January 2021).
- CheckMarx Software Official Website. Available online: <https://www.checkmarx.com> (accessed on 19 January 2021).
- 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.
- 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.
- Moshtari, S.; Sami, A. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016.
- Shin, Y.; Williams, L. Can traditional fault prediction models be used for vulnerability prediction? *Empir. Softw. Eng.* **2013**, *18*, 25–59. [\[CrossRef\]](#)
- 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. [\[CrossRef\]](#)
- Morrison, P.; Herzig, K.; Murphy, B.; Williams, L. Challenges with applying vulnerability prediction models. In Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, Urbana, IL, USA, 21–22 April 2015.
- 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, 3–7 September 2016.
- 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. [\[CrossRef\]](#)
- Riccardo, S.; James, W.; Aram, H.; Wouter, J. Predicting vulnerable software components via text mining. *IEEE Trans. Softw. Eng.* **2014**, *40*, 993–1006.
- Pang, Y.; Xue, X.; Namin, A.S. Predicting vulnerable software components through n-gram analysis and statistical feature selection. 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA, 9–11 December 2015.
- 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.
- Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secure Comput.* **2021**, *1*. [\[CrossRef\]](#)
- Li, X.; Wang, L.; Xin, Y.; Yang, Y.; Chen, Y. Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning. *Appl. Sci.* **2021**, *10*, 1692. [\[CrossRef\]](#)
- 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. [\[CrossRef\]](#)
- 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.
- Bilgin, Z.; Ersoy, M.A.; Soykan, E.U.; Tomur, E.; Çomak, P.; Karaçay, L. Vulnerability Prediction from Source Code Using Machine Learning. *IEEE Access* **2020**, *8*, 150672–150684. [\[CrossRef\]](#)
- Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. Code2vec: Learning distributed representations of code. In Proceedings of the ACM on Programming Languages (POPL), Cascais, Portugal, 14–15 January 2019.
- Alon, U.; Brody, S.; Levy, O.; Yahav, E. Code2seq: Generating Sequences from Structured Representations of Code. In Proceedings of the International Conference on Learning Representations, New Orleans, LA, USA, 6–9 May 2019.
- Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
- Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. 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), Montréal, QC, Canada, 25–31 May 2019.
- Tufano, M.; Watson, C.; Bavota, G.; Di Penta, M.; White, M.; Poshvanyk, D. Deep learning similarities from different representations of source code. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden, 27 May–3 June 2018.
- Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. In Proceedings of the 6th International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
- 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.

27. Duan, X.; Wu, J.; Ji, S.; Rui, Z.; Luo, T.; Yang, M.; Wu, Y. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, Macao, China, 10–16 August 2019.
28. 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.
29. Moshtari, S.; Sami, A.; Azimi, M. Using complexity metrics to improve software security. *Comput. Fraud. Secur.* **2013**, *5*, 8–17. [\[CrossRef\]](#)
30. Hovsepian, A.; Scandariato, R.; Joosen, W. Is Newer Always Better? The Case of Vulnerability Prediction Models. In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Ciudad Real, Spain, 8–9 September 2016.
31. Shin, Y.; Williams, L. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In Proceedings of the 7th International Workshop on Software Engineering for Secure Systems, Waikiki, Honolulu, HI, USA, 22 May 2011.
32. Du, X.; Chen, B.; Li, Y.; Guo, J.; Zhou, Y.; Liu, Y.; Jiang, Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019.
33. 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.
34. Yamaguchi, F.; Wressnegger, C.; Gascon, H.; Rieck, K. Chucky: Exposing missing checks in source code for vulnerability discovery. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013.
35. 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. [\[CrossRef\]](#)
36. Dam, H.K.; Tran, T.; Pham, T.T.M.; Ng, S.W.; Ghose, A. Automatic feature learning for predicting vulnerable software components. *IEEE Trans. Softw. Eng.* **2021**, *47*, 67–85. [\[CrossRef\]](#)
37. 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, 18–20 May 2016.
38. Liu, S.; Lin, G.; Qu, L.; Zhang, J.; De Vel, O.; Montague, P.; Xiang, Y. CD-VulD: Cross-Domain Vulnerability Discovery based on Deep Domain Adaptation. *IEEE Trans. Dependable Secure Comput.* **2020**, PrePrints. [\[CrossRef\]](#)
39. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; De Vel, O.; Montague, P.; Xiang, Y. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Trans. Dependable Secure Comput.* **2019**, Early access. [\[CrossRef\]](#)
40. Li, Z.; Zou, D.; Tang, J.; Zhang, Z.; Sun, M.; Jin, H. A comparative study of deep learning-based vulnerability detection system. *IEEE Access* **2019**, *7*, 103184–103197. [\[CrossRef\]](#)
41. Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; Jin, H. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *arXiv* **2020**, arXiv:2001.02350.
42. Cho, K.; Merriënboer, B.V.; Gulcehre, C.; Ba Hdanau, D.; Bougares, F.; Schwenk, H. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In Proceedings of the 2014 Conference on Empirical Methods on Natural Language Processing, Doha, Qatar, 26–28 October 2014.
43. SARD Manual. Available online: <https://samate.nist.gov/index.php/SARD.html> (accessed on 28 December 2020).
44. Clang: A C Language Family Frontend for LLVM. Available online: <https://clang.llvm.org/> (accessed on 28 December 2020).
45. Introduction of DG. Available online: <https://github.com/mchalupa/dg> (accessed on 28 December 2020).
46. Flawfinder Software Official Website. Available online: <https://www.dwheeler.com/flawfinder/> (accessed on 29 December 2020).