algorithms

Article

A Context-Aware Neural Embedding for Function-Level Vulnerability Detection

Hongwei Wei [1], Guanjun Lin [1,*], Lin Li [2] and Heming Jia [1,*]

[1] School of Information Engineering, Sanming University, Sanming 365004, China; 20180861226@fjsmu.edu.cn
[2] School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne 3122, Australia; linli@swin.edu.au
* Correspondence: guanjun.lin@fjsmu.edu.cn (G.L.); jiaheming@fjsmu.edu.cn (H.J.)

Abstract: Exploitable vulnerabilities in software systems are major security concerns. To date, machine learning (ML) based solutions have been proposed to automate and accelerate the detection of vulnerabilities. Most ML techniques aim to isolate a unit of source code, be it a line or a function, as being vulnerable. We argue that a code segment is vulnerable if it exists in certain semantic contexts, such as the control flow and data flow; therefore, it is important for the detection to be context aware. In this paper, we evaluate the performance of mainstream word embedding techniques in the scenario of software vulnerability detection. Based on the evaluation, we propose a supervised framework leveraging pre-trained context-aware embeddings from language models (ELMo) to capture deep contextual representations, further summarized by a bidirectional long short-term memory (Bi-LSTM) layer for learning long-range code dependency. The framework takes directly a source code function as an input and produces corresponding function embeddings, which can be treated as feature sets for conventional ML classifiers. Experimental results showed that the proposed framework yielded the best performance in its downstream detection tasks. Using the feature representations generated by our framework, random forest and support vector machine outperformed four baseline systems on our data sets, demonstrating that the framework incorporated with ELMo can effectively capture the vulnerable data flow patterns and facilitate the vulnerability detection task.

Keywords: code neural embedding; contextual learning; vulnerability detection

check for updates

Citation: Wei, H.; Lin, G.; Li, L.; Jia, H. A Context-Aware Neural Embedding for Function-Level Vulnerability Detection. *Algorithms* **2021**, *14*, 335. https://doi.org/10.3390/a14110335

Academic Editor: Laurent Risser

Received: 5 October 2021
Accepted: 15 November 2021
Published: 17 November 2021

1. Introduction

Recently, the rapid increase in the number of disclosed software vulnerabilities has posed a huge security threat to the cyberspace worldwide [1–5]. To combat the potential cyber threats caused by exploitable vulnerabilities in software, machine learning (ML) and data-driven based approaches have been proposed for bug/vulnerability detection [6–8]. Recent studies based on conventional ML and deep learning (DL) are systematically reviewed in these survey papers (i.e., [9–12]), indicating that the data-driven approaches can be alternative yet effective solutions to assist code inspection. Particularly, the emerging DL techniques, which can perform automated feature extraction, have relieved practitioners from tedious and potentially error-prone feature engineering tasks [5,13]. Provided with enough data, feature representations automatically learned by DL algorithms can be more effective than the ones generated by human experiences and possibly with an improved level of generalization ability [5,14].

The application of deep learning for code analysis requires the transformation of software code (i.e., the source code) to vector representations recognizable by DL algorithms; however, this is still challenging [15]. Effective vector representation ensures that the semantics and syntax of the software code are preserved, better facilitate the learning process and eventually benefit downstream code analysis tasks. Due to the bimodal property of software code (i.e., being understandable by machines and readable by developers), many natural language processing (NLP) techniques have been applied for code processing [10,16,17]. Source code can be encoded at the token level [18,19], or at character level [20,21], and processed as text. Nevertheless, these techniques generally represent individual words as atomic units, ignoring the similarity between words and the relationship among them, thus causing difficulties for the downstream algorithms to learn expressive and rich semantics related to context. These issues were tackled by distributed word embedding techniques, such as Word2Vec [22], GloVe (global vectors for word representation) [23], and FastText [24]. For example, techniques such as Word2Vec can learn word probability based on contextual information, are capable of capturing word similarities, and form the foundation for many existing studies that require the learning of code semantics for code analysis tasks, such as vulnerability detection [5,14,25–28].

Software vulnerability detection is a context-sensitive code analysis task, often requiring the analysis of a broad code context to better track the data flows and/or control flows, or even to understand data dependencies [5,10,14]. Therefore, the contextual information of source code tokens must be captured accurately. When we apply the distributed word embedding techniques (e.g., Word2Vec) for vulnerability detection, they can only learn word embeddings based on a small context window, and fail to capture the meaning that is interdependent between the target words and their context as a whole [29]. Additionally, Word2Vec, which is a non-contextualized embedding method, can only generate one embedding for one word. If a word has multiple meanings, Word2Vec cannot produce different embeddings for that word based on its contexts. The inability to generate different embeddings based on the difference of contexts may result in the incorrect or incomplete representations of code contexts and eventually affect the performance of downstream code analysis tasks.

In this paper, we propose a code-embedding framework based on the embeddings for language models (ELMo) [15,30] to facilitate the learning of contextual code semantics. The ELMo is trained on the 1 billion word benchmark [31], capable of capturing `deep contextualized` word representations [30] and generating different embeddings according to different contexts of a word. It can also be fine-tuned on domain-specific data [15], which means that domain transfer can be achieved. We evaluate five mainstream word embedding techniques, demonstrating that the code representations produced by the ELMo module outperformed the selected embedding models on our software source code data sets for vulnerability discovery.

The structure of our proposed framework consists of the ELMo module followed by a bidirectional long short-term memory (Bi-LSTM) network for converting source code functions to vector representations indicative of vulnerable patterns. The proposed framework inherits the capabilities of the ELMo model, being able to learn meaningful representations of source code tokens based on code semantics. It is also capable of capturing long-term contextual dependencies associated with the source-sink patterns, which is the key for vulnerable patterns recognition. Experiments showed that our framework could be easily integrated with many conventional ML algorithms and is capable of capturing vulnerable semantics for detecting vulnerabilities. In summary, our contributions are three-fold:

- We compare the performance of five mainstream word embedding models, including Word2Vec, GloVe, FastText, ELMo, and bidirectional encoder representations from transformers (BERT) [32]) in the scenario of vulnerable function detection. We demonstrate that among the selected NLP embedding solutions, ELMo is the most suitable one for generating code embedding for vulnerability detection when there are not many code samples available for pre-training.
- Based on the evaluation of embedding solutions, we design a deep neural network built on top of the ELMo module to capture code semantic that is context aware and is capable of learning the long-term dependencies, reflecting potentially vulnerable source-sink patterns in the source code.
- On top of the ELMo-based neural model, we develop a framework for transforming source code functions to meaningful embeddings optimized for vulnerable function detection, without the need for other code analysis tools. A performance evaluation is carried out

to examine the effectiveness of the proposed framework with four baseline systems. The results confirm that the proposed framework achieved state-of-the-art performance.

The rest of this paper is organized as follows: Section 2 describes how the code-embedding framework for vulnerability detection is designed for capturing the vulnerable patterns in details. The experiments for the evaluation of the proposed framework are presented in Section 3. Section 4 lists the related studies, and Section 5 concludes the paper.

2. Framework Design

This section presents the workflow of the proposed framework and how it is designed to capture the potentially vulnerable code context within the function boundary.

2.1. Workflow

The framework we propose is a supervised solution for extracting function-level embeddings, specifically designed for vulnerable function discovery. The framework is a deep neural network and is designed for the scenario where there are some identified vulnerable data available for a software project. As Figure 1 shows, the workflow consists of three stages. In the first stage, we train the network with source code functions that are labeled as vulnerable or non-vulnerable. This allows the built-in ELMo module to be fine-tuned by the software code data. In the second stage, we feed both the labeled and unlabeled functions to the trained network and extract one of the hidden layers' outputs (in this paper, we use the third last layer's output) as the learned embeddings. Since the framework takes a source code function as an input, the output embedding can be seen as a distributed representation of the corresponding input function, which is related to the code semantics and syntax indicative of vulnerable code patterns. In the last stage, the extracted function embeddings can be used as feature vectors for ML classifiers for further classification.

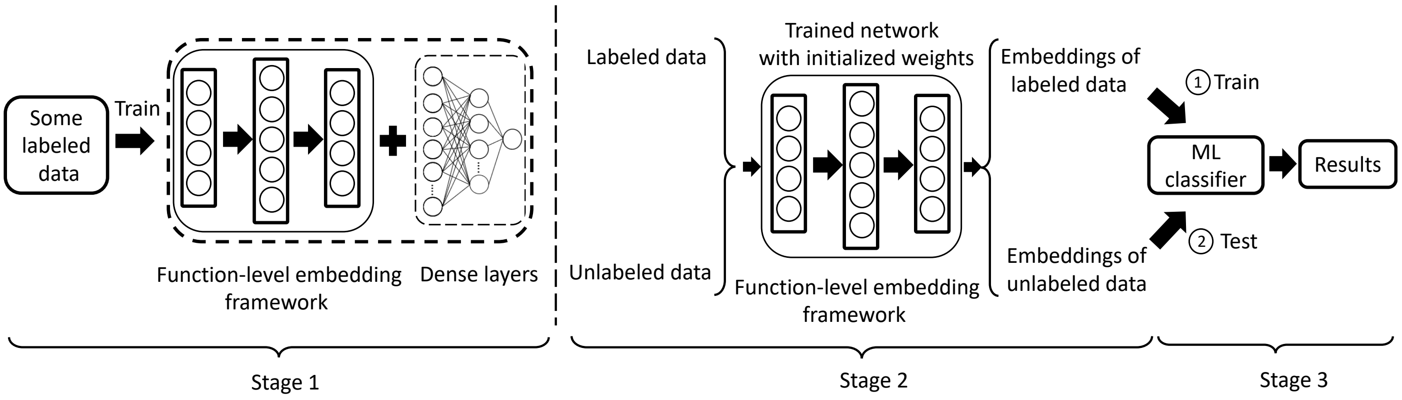
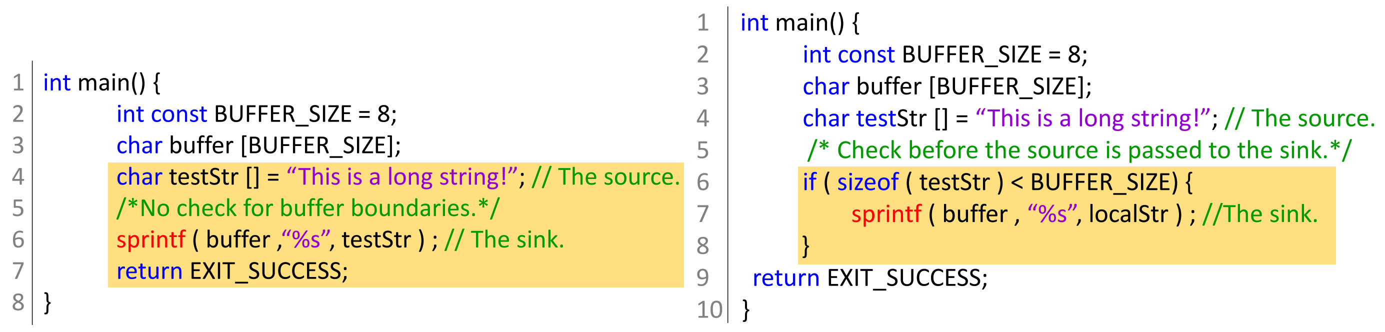

Some labeled data
Train
Function-level embedding framework
Dense layers
Labeled data
Unlabeled data
Trained network with initialized weights
Function-level embedding framework
Embeddings of labeled data
Embeddings of unlabeled data
① Train
② Test
ML classifier
Results
Stage 1
Stage 2
Stage 3

Figure 1. The proposed framework is designed for the scenario where there are some labeled data for a software project. This scenario consists of three stages: the first stage is to train the neural network with the labeled data; the second stage is to feed both the labeled and the unlabeled data to the trained network to obtain one of the hidden layers' output as the learned function-level embeddings; and in stage three, the resulting embeddings can be used as feature sets for subsequent process, e.g., to train a conventional ML classifier for further classification.

2.2. Code Context Analysis

To capture the vulnerable programming patterns that lead to vulnerabilities, it is crucial to analyze the code context. A typical vulnerable pattern can be that when a variable flows from an appropriate source to a corresponding sink, which does not go through proper validation (e.g., buffer size check). Many different types of vulnerabilities share such a pattern [33]. As it is shown by the code examples in Figure 2, it depicts a vulnerable context: the source, which is a local variable `testStr` (at line 4) is assigned with a string whose length exceeds the size of `buffer` which is defined at line 3. When `testStr` is passed to the sink sprintf at line 6, there is no size check of variable `testStr`, which leads to a buffer overflow vulnerability.



```
1  int main() {
2          int const BUFFER_SIZE = 8;
3          char buffer [BUFFER_SIZE];
4          char testStr [] = “This is a long string!”; // The source.
5          /*No check for buffer boundaries.*/
6          sprintf ( buffer ,“%s”, testStr ) ; // The sink.
7          return EXIT_SUCCESS;
8  }
```

(**a**)

```
1  int main() {
2          int const BUFFER_SIZE = 8;
3          char buffer [BUFFER_SIZE];
4          char testStr [] = “This is a long string!”; // The source.
5           /* Check before the source is passed to the sink.*/
6          if ( sizeof ( testStr ) < BUFFER_SIZE) {
7                sprintf ( buffer , “%s”, localStr ) ; //The sink.
8          }
9    return EXIT_SUCCESS;
10 }
```

(**b**)

Figure 2. The different contexts of the target token `sprintf` (red). (**a**) The potentially vulnerable context (yellow) of target token (red): `sprintf`. (**b**) The patched context (yellow) of target token (red): `sprintf`.

To identify such a vulnerable code context that spans across multiple statements from a source variable to a potentially vulnerable sink, a detection method should be able to (1) identify the potentially vulnerable context, which consists of at least a source and a sink, and (2) recognize that there has been no proper validation prior to passing the source variable to the sink. To achieve these goals, a method should be able to learn the semantics of code tokens and capture the contextual dependencies that range from source variable declaration/assignment to a sink function invocation. Our embedding framework adopts a neural network consisting of multiple layers to learn both code semantics and long-term contextual dependencies.

2.3. Contextual Semantic Learning

The key to identify the potentially vulnerable context is to learn accurate code semantics that correctly represents its contextual meaning. We can add an `if` statement before the sensitive sink `sprintf` is called, and the `if` statement checks the size of the variable `testStr` (see Figure 2b code sample at line 6). This new code piece would remedy the vulnerability because if the size of variable `testStr` exceeds the array size, the sink will not be called. Therefore, whether the sink `sprintf` is vulnerable depends on whether its preceding code context has validation statements or not. Hence, the same token `sprintf` in different contexts (with or without validation of its parameters) should be represented differently, and being able to recognize these differences is the key to detect vulnerable patterns.

The techniques such as Word2Vec capture the meaning of a token based on a relatively small context window. Therefore, we applied the ELMo to produce word representations by learning the representation of each word based on a `deep context`. The deep context is a broader context learned by combining the internal states of the bidirectional LSTM. This feature enables the ELMo to produce different representations of a word based on the different contexts in which the word is used [15]. For instance, considering the sensitive sink `sprintf` as the target token shown in Figure 2, ideally, the ELMo should be able to capture both the vulnerable source-sink code context as described in Figure 2a, and the patched source-validation-sink context as shown in Figure 2b.

To evaluate whether the ELMo can generate different embeddings for a target sink function based on the difference contexts, we constructed three potentially vulnerable functions which do not have any validation for the parameters and three patched versions with *if* statements for validation of the parameters prior to the functions being called. Figure 3 shows the 2D plot of these function embeddings learned by the ELMo. The t-SNE algorithm was used to map the 1024-dimensional vector representations to a 2D plane. It can be seen that the learned representations of the vulnerable functions represented by the red rectangles are separated with that of the corresponding patched versions, which have *if* statements for validation. It proves that the contextual information (i.e., with or without validation) of the code can be learned by the ELMo. This will facilitate the analysis for vulnerability detection.

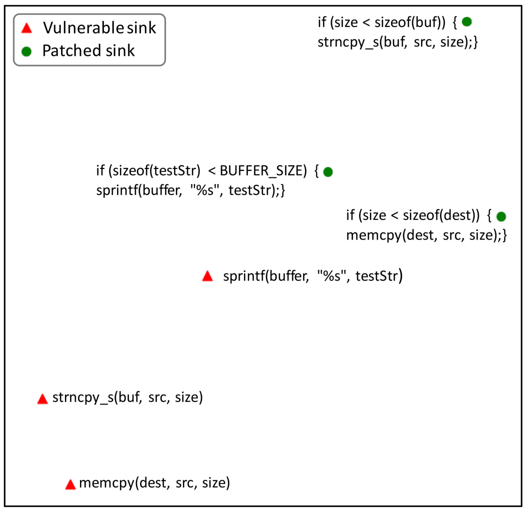


Figure 3. A plot of the ELMo representations of three vulnerable sinks and their corresponding patched versions. The representations are 1024-dimensional vectors output from the mean-pooling layer of the ELMo module, which are projected onto a 2D plane using t-SNE (with perplexity = 2.2). The red rectangles are the representations of the vulnerable sinks missing proper validation. The green circles are the representations of the patched sinks using *if* statement for validation. The figure depicts that the ELMo module could produce distinct representations for vulnerable sinks and non-vulnerable ones based on their context (i.e., having *if* statement for validation or not.)

2.4. Handling Out-of-Vocabulary Words

When code tokens are converted to meaningful embeddings, the words that do not appear in the training set, called out-of-vocabulary (OOV) words, may result in information loss. Compared with Word2vec, GloVe, FastText, and BERD, ELMo embedding is naturally constructed word representation at the character level, which allows robust embeddings to be produced for OOV words unseen in the training [30]. The capability of handling OOV words is important for learning robust and generalizable representations for software source code. One way in which the source code differs from natural language is that the function names, variable/parameter names from different software projects, usually do not follow the same naming conventions/rules. For instance, a string type variable can be named `testStr` in one function, and be called `str` in another function written by a different programmer. Both `testStr` and `str` tokens can be nonexistent on the code base used for training. Therefore, applying the embedding mechanisms, such as Word2Vec, would result in the unseen tokens being omitted or replaced by a special token (e.g., OOV), eventually causing information loss.

2.5. Long-Term Dependency Learning

Another key for detecting the vulnerable patterns is to capture the long-term dependencies. In an ANSI C function, a data flow which starts from the declaration of a local variable (which is the source) to the corresponding sink can consist of multiple statements spanning across the whole function body. To recognize the potentially vulnerable patterns manifested in the data flow, our embedding framework implements a one-layer bidirectional LSTM (Bi-LSTM) network followed by the ELMo embedding to ensure the learning of long-term contextual dependencies.

The LSTM implementation was introduced to handle long-term dependencies of input sequences. Compared to a vanilla recurrent neural network (RNN) or the gated recurrent unit (GRU), which is another variant of an RNN, the LSTM performs better at tracking long-term dependencies, due to having both the gates and a memory cell for keeping/updating states from the previous and the next candidate activations. As mentioned previously, a potentially vulnerable data flow usually encompasses a number of code statements, either preceding or subsequent, or even both. Therefore, the bidirectional LSTM architecture is implemented to enhance the ability to learn the long-range dependencies in both forward and reverse directions, which can effectively capture the vulnerable data flow that is context dependent.

2.6. Framework Implementation

The proposed function-level embedding framework is illustrated in Figure 4, and is explained, as follows, in detail.

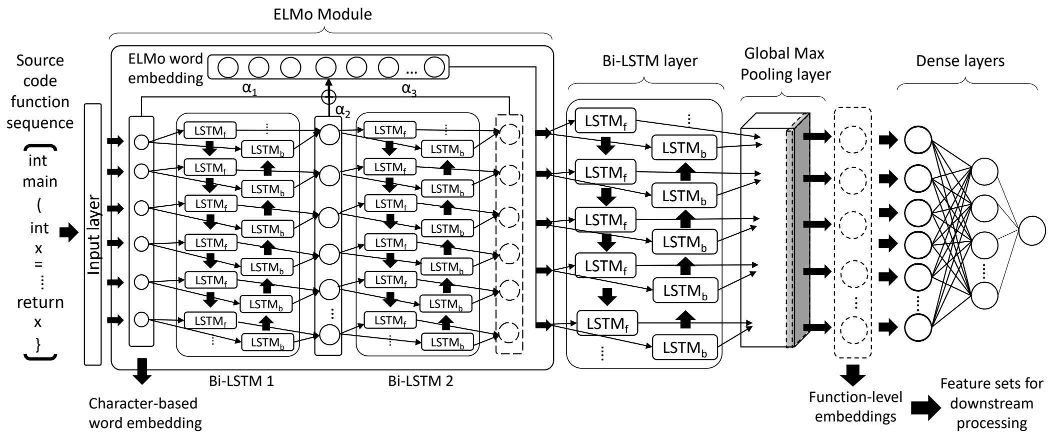


Figure 4. The structure of the proposed deep neural network with the ELMo module built-in for extracting function-level embeddings. The ELMo module exposes 3 trainable weights—α_1, α_2 and α_3—for the aggregation of the character-based word embedding layer, and the two Bi-LSTM layers, respectively. We remove the mean-pooling layer and connect the ELMo word embedding layer to another Bi-LSTM layer, followed by a global max pooling layer to form the embedding framework. During the training phase, three dense layers are added to the global max pooling layer to train the network. When training is completed, the source code functions can be fed to the input layer and the embeddings are the output of the third last layer (global max pooling layer).

2.6.1. Input Preparation

The framework takes textual sequences of source code functions as input, and therefore, no program analysis tools are needed. Each input sequence corresponds to a source code function, which is in a serialized form. Before feeding the sequences to the network, we need to convert sequences to a constant input length. Due to the overly long sequences degrading the Bi-LSTM performance, we have to balance the length and the sparsity of the input sequences. We choose 1000 as the sequence length threshold, considering that the 90% of sequences have fewer than 1000 elements. For long sequences, we truncate their length to 1000, and short ones are padded with special characters at the end of each sequence. We are aware that the truncation to the overly long sequences may result in the actual vulnerable code, which is not in the preceding 1000 elements being removed. This issue is discussed in Section 3.5.

2.6.2. Network Architecture

After the sequences are truncated to the unified length, they are ready to be fed to the network. As Figure 4 shows, the ELMo module followed by the input layer treats each source code function (untokenized) as a sentence. The ELMo module integrated in our network is provided by Tensorflow Hub (https://tfhub.dev/google/elmo/2, accessed on 10 November 2021). It is a 5-layer neural network, which exposes three trainable parameters for the aggregation of the first three layers. We remove the mean-pooling layer so that the fourth layer, which is the ELMo word embedding layer, outputs the word embeddings acceptable by the subsequent Bi-LSTM layer. The Bi-LSTM layer contains the 64 forward and 64 backward LSTM units, respectively, forming a bi-directional structure. Then, outputs of the forward and backward LSTM units are concatenated and passed to a pooling layer. To speed up the Bi-LSTM training process, we use the CudnnLSTM implementation, which is the LSTM based on the NVIDIA CUDA deep neural network library (https://developer.nvidia.com/cudnn, accessed on 10 November 2021).

The pooling layer is the last layer of the proposed framework, which aims to reduce the output dimensionality to a dense vector of a fixed size, acceptable by mainstream ML algorithms. In this paper, we use the global max pooling layer to retain the most important information as represented by the maximal value of the activations from the Bi-LSTM layer. Since there is at most one vulnerability in each sample in our data sets, we surmise that the vulnerable elements in the input sequence will result in having large values after the processing of the preceding layers. Applying the global max pooling for selecting the maximum value can, thus, help to identify potentially vulnerable signals.

2.6.3. Function-Level Embedding Generation

The framework needs to be trained to generate function-level embeddings for vulnerability detection, as shown in Figure 1. During the training phase, the global max pooling layer is connected to three dense layers to form a deep neural network as Figure 4 depicts. We also add a dropout (with a value of 0.2) layer before the dense layers to prevent overfitting. The optimizer we use is the stochastic gradient descent (SGD) with gradient clipping mechanism to guarantee that the network converges properly; the loss function to minimize is the binary cross-entropy. We use a relatively small batch size (with a value of 8) for training the network, as, proved by [34], a small batch size generally improves the quality of the model in terms of generalization. As a trade-off, we train the network with relatively large epochs (150).

The training process aims to fine-tune the parameters of the ELMo embedding layer and initialize the weights of the followed Bi-LSTM layer to differentiate the potentially vulnerable functions from non-vulnerable ones. We assume that the labeled data available for training the network are very limited. Therefore, we do not directly use the trained network as a classifier for the detection of vulnerable functions. When training is completed, the dense layers are removed. We use the output of the global max pooling layer as the learned function-level embeddings ready for vulnerable discovery.

3. Experiments and Evaluation

This section describes the real-world case studies using our proposed framework for evaluating the effectiveness of the extracted embeddings for vulnerable code function detection.

3.1. Experiment Data Sets

The proposed function-level embedding framework was evaluated on the real-world vulnerability data set collected by our previous work Cross-VD [14]. This data set contains manually labeled vulnerable functions and collected non-vulnerable functions from three popular C open source software projects, as listed in Table 1. According to [14], the vulnerability labels were obtained from the National Vulnerability Database (NVD) [35] and the Common Vulnerability and Exposures (CVE) [36] websites. To match the labels with the source code function, the corresponding versions of a project's source code were downloaded from GitHub. Then, each vulnerable function in the software project was located based on the information provided by NVD and CVE websites and manually labeled. For the vulnerabilities that spanned across functions or across files (e.g., interprocedural vulnerabilities), they discarded them. Excluding the identified vulnerable functions and the discarded vulnerabilities, the authors [14] used the remaining functions as the non-vulnerable ones. In this paper, we only chose two open source projects that contain the most number of vulnerable functions, which are FFmpeg and LibTIFF. We also manually labeled another project OpenSSL based on their method.

The real-world vulnerability data sets aim to test the performance of our framework on practical situations, where there are insufficient labeled data available for training. To perform the experiments, we partitioned the data sets collected from each project as Table 2 shows.

Table 1. The data set used in the experiments.

Data Source	Dataset	# of Functions Used/Collected	
		Vulnerable	**Non-Vulnerable**
Open-source projects	FFmpeg	213	5701
	LibTIFF	96	731
	OpenSSL	143	7068

Table 2. The training/test partitions of the data sets. We set 33% of the training samples to simulate the case where there is a limited number of labeled vulnerability data.

Data Set	Training Set		Test Set	
	# of Vulnerable Functions	**# of Non-Vulnerable Functions**	**# of Vulnerable Functions**	**# of Non-Vulnerable Functions**
FFmpeg	70	1881	143	3820
LibTIFF	32	241	64	490
OpenSSL	47	2332	96	4736

3.2. Experiment Settings

The design and the effectiveness of the proposed framework are evaluated on real-world vulnerability data sets, aiming to examine whether the generated embeddings are effective for vulnerable function classification. Experiments are conducted on the three open-source projects: FFmpeg, LibTIFF, and OpenSSL. To simulate the scenario that there are usually insufficient labeled vulnerable functions available, we partition the data sets into two parts: 33% training and 67% test, as shown in Table 2. We keep the ratio of training and test partition unchanged in all our experiments. To ensure the validity of reported results, we partition the data set of each project three times to generate three different training and test sets for each run. The final results we obtain are the average of the results from three runs.

The data imbalance issue is addressed using the cost-sensitive learning. In practice, the number of vulnerable functions is significantly smaller than the non-vulnerable ones in real-world projects, which may bias the learning of neural models. Therefore, during the training phase in Stage 1 (shown in Figure 1), cost-sensitive learning is applied by assigning different weights to vulnerable and non-vulnerable classes, so the loss function is adjusted to balance two imbalanced classes. In this paper, the following equation is used to obtain the weights of each class:

$$class\_weight = \frac{total\_samples}{n\_classes * one\_class\_samples} \tag{1}$$

where the value of $n\_classes$ is 2, denoting the two classes which are vulnerable functions and non-vulnerable ones. The $one\_class\_samples$ refers to the number of samples in one class. During the training phase, the misclassification cost of the samples in the vulnerable class is higher than that of non-vulnerable ones, enabling the neural network to overcome the data imbalance issue.

3.2.1. Settings for Evaluating the Effectiveness of Embedding Models

The first group of experiments aims to examine whether ELMo is more suitable than other mainstream word embedding models (i.e., Word2Vec, GloVe, FastText, and BERT) in terms of learning meaningful representations from the source code for the downstream vulnerability detection tasks. If the ELMo can learn contextual semantics of code more effectively than other mainstream word embedding models for vulnerability detection, the RF trained by the embeddings generated by ELMo should achieve better results, compared with the embeddings generated by other models.

We apply different pre-training and fine-tuning strategies for selected embedding models. For the non-contextualized embedding models, which are Word2Vec, FastText, and GloVe, we pre-train them using all the code samples from three open-source projects because their pre-training process is unsupervised. For contextualized embedding models, which are EMLo and BERT, we perform fine-tuning using the training sets (listed in Table 2) since they are pre-trained. To perform evaluations, we use source code functions as inputs to ELMo and other embedding models, respectively. The outputs are generated code representations which are treated as feature sets. Then, these generated representations are directly inputted to the RF for training and test.

3.2.2. Settings for Determining the Structure of Bi-LSTM

The second group of experiments is to examine how many layers of Bi-LSTM are suitable for capturing the long-term dependency that leads to better classification results in terms of vulnerability detection. For each open source project, we train different networks consisting of different number of Bi-LSTM layers to investigate how results vary with the number of Bi-LSTM layers changing from 0 to 3. Then, we use the network structure that achieved the best performance for further evaluation.

3.2.3. Settings for Evaluating on Real-World Open Source Projects

The third group of experiments are to compare our framework with four function-level vulnerability detection systems in terms of their performance on real-world software projects. For comparison, we choose one open-source static code analysis tool which is Flawfinder (version 2.0.7) [37] and three DL-based approaches which are Cross-VD [14], Multi-VD [27], and DeepBalance [38] as the baselines.

The choice of the baseline systems is due to the fact that Flawfinder was used in many previous studies as a recognized benchmark, such as [5,6]. It takes a source code file/function as input and outputs a line number, indicating the location of possible security flaws with the level of severity. It also generates warnings, suggestions and flaw types to facilitate code fixes. Flawfinder is a pattern-based code scanner with well-known vulnerable patterns built-in. By comparing the code pattern with the pre-defined vulnerable patterns, the tool raises warnings when the code pattern matches the security flaw pattern.

Cross-VD [14] builds on a two-layer Bi-LSTM network with Word2Vec embedding. Their idea is to learn representations from historical real-world vulnerability data sets and used the generated representations from the Bi-LSTM as feature sets for training a random forest classifier. Instead of using source code functions, their work takes abstract syntax trees (ASTs) extracted from source code functions as inputs and uses the generated representations from the Bi-LSTM as feature sets for training a random forest classifier for vulnerable function detection.

Multi-VD [27] utilizes two Bi-LSTM networks to extract representations from both the artificially constructed vulnerability data source and real-world open-source projects. Therefore, one network is fed with the ASTs extracted from source code functions derived from real-world open-source projects. The other network is fed with the source code of the artificially constructed function samples. Then, the extracted representations from two data sources, which are generated by the two networks, are concatenated and fed to a random forest classifier for further training and testing.

DeepBalance [38] aims to deal with the data imbalance issue in the scenario of vulnerability detection. The authors also utilize the high-level representations extracted from a Bi-LSTM neural network as features for training and apply a fuzzy oversampling technique to rebalance the training data by generating synthetic samples for the vulnerable class. DeepBalance also takes ASTs as inputs and performs fuzzy-based class rebalancing on AST-based feature representations.

In this group of experiments, we use the embeddings generated from the proposed framework as feature sets and feed them to conventional ML algorithms: a random forest (RF) algorithm and a support vector machine (SVM) with radial basis function kernel for the classification of potential vulnerabilities.

3.2.4. Settings for the Detection of Context-Related Vulnerabilities

The fourth group of experiments are to evaluate whether the proposed framework which utilizes a Bi-LSTM network incorporated with ELMo could facilitate the analysis of the code context and the detection of context-related vulnerabilities. We examine how many among the identified vulnerabilities are actually context-related. If there are higher proportions of context-related vulnerabilities in the found vulnerabilities in the top-k retrieved function list compared with the proportion of context-related vulnerabilities in the test set, it means that the proposed framework can contribute to the detection of vulnerable contexts.

We choose the type of buffer errors (CWE-119) (the CWE-119 is also known as the improper restriction of operations within the bounds of a memory buffer) vulnerabilities as the target context-related vulnerabilities for the evaluation. The buffer errors vulnerability type includes out-of-bounds read (CWE-125), out-of-bounds write (CWE0-787), and many other sub-types related to improper manipulation of buffers in memory, which require the analysis of code contexts usually containing a key variable and a potentially vulnerable sink. Due to missing bounds check, specially crafted data held by the variable can cause a buffer error when it is passed to the vulnerable sink [33].

When partitioning the data set into training and test sets, we keep the proportion of CWE-119 vulnerabilities roughly equal in both sets. Then, we examine among the top-k retrieved function lists whether the proportion of CWE-119 vulnerabilities would be higher than the proportion in the test set.

3.2.5. Experiment Environment

The proposed framework and the baseline approach [14] were implemented using *Keras* (version 2.2.4) [39] and *TensorFlow* (version 1.14.0) [40]. The RF and SVM algorithms were sourced from the *scikit-learn* package (version 0.20.0) [41]. The software tool for implementing Word2Vec and FastText embeddings were implemented using the *gensim* package (version 3.4.0) [42] using all default settings. The Python implementation of GloVe was based a repository (https://github.com/maciejkula/glove-python, accessed on 10 November 2021) on GitHub. The computational system used was a desktop running Windows 10 with 32GB RAM and an NVIDIA TITAN Xp GPU.

3.3. Evaluation Criteria

To evaluate the performance of classification accuracy, we apply ranked retrieval measurement due to the highly imbalanced classes of vulnerable and non-vulnerable functions. We retrieve k functions, which are classified as the most probable vulnerable functions, and calculate the proportion of the actual vulnerable functions returned in the list of k functions. Formally, this measure can be formulated as top-k precision (denoted as $P@K$) and top-k recall (denoted as $R@K$). For the vulnerable class, $P@K$ and $R@K$ can be calculated using the following equations:

$$P@K = \frac{TP@k}{TP@k + FP@k}, \; R@K = \frac{TP@k}{TP@k + FN@k} \tag{2}$$

P@K denotes the proportion in the top-*k* retrieved functions that the actual vulnerable functions account for, and the *R@K* is the proportion of vulnerable functions that are in the top-*k* retrieved functions.

In practice, these metrics are generally used in the context of information retrieval systems (e.g., search engines) for measuring how many relevant documents are retrieved in all the top-*k* retrieved documents [43], where the relevant documents account for a small portion of the whole document set. Similarly, in our context, the number of vulnerable functions is considerably smaller in number than non-vulnerable ones, and in practice, the number of functions retrieved is bounded in a situation where not all code is able to be audited, due to time and resource limitations. For Flawfinder, we use the results that have a risk level larger or equal to 1 and rank the results based on the risk level in descending order.

3.4. Result Analysis

3.4.1. The Effectiveness of Embedding Models

To explore whether the ELMo module is more effective than other mainstream word embedding models in terms of learning contextual code semantics, we compare the effectiveness of code representations generated by these five embedding solutions by training a vulnerable function classifier, using the generated representations as features. The ELMo module takes a textual vector as input and by default, it outputs a fixed size numeric vector as the embedding of the corresponding input which can be directly fed to a ML classifier. The Word2Vec model takes a numeric vector and outputs a fixed size vector for each element in the input vector, resulting in a two-dimensional array as the learned embeddings. GloVe takes advantage of all the information in the corpus by learning the word co-occurrence matrix of the corpus. FastText utilizes sub-words to learn the expression of words, each word consisting of an internal string of n-gram letters. After introducing the factor of sub-words, the micro-deformation relationship of words can also be mapped into the embedded space. BERT is a representative pre-training model based on the bidirectional transformer structure. Different from ELMo, which uses Bi-LSTM to capture word semantics, BERT applies the transformer's encoder for obtaining representations of words. With the multi-attention mechanism, BERT has the potential of better understanding of code semantics and syntax. GloVe, FastText and BERT also can accept a word as input and enter a fixed-size word embedding vector for the corresponding word. Then, we apply an average pooling for converting the 2D array to vectors and then feed them to a ML classifier.

Table 3 shows the results of RF using two groups of embeddings generated by the ELMo and other mainstream word embedding models as feature sets on three projects: FFmpeg, LibTIFF, and OpenSSL. Generally, RF using the ELMo generated embeddings produces better results than using the embeddings generated by other mainstream word embedding models on all three projects. When retrieving fewer than 30 functions, the performance gap of using five groups of embeddings is not distinct. With more functions returned, the classifier using the ELMo embedding could identify significantly more vulnerable functions, especially for project FFmpeg, and OpenSSL.

Even the state-of-the-art word embedding model BERT does not perform as well as our ELMo-based framework. The underlying reasons are two-fold: (1) the BERT model we applied is not trained on a large amount of C source code, and (2) the word-piece (WPE) mechanism used for training BERT on natural language may bias the model when it is used for code analysis tasks. The BERT model not being trained in the C programming language is the root cause of the performance degradation. Unlike natural languages that have fixed vocabularies and phrases, the names of variables and functions in source code can vary because software projects usually have their own naming conventions. In a scenario of a NLP task, where WPE is applied for handling the OOV problem, words such as "tester", "tested", and "testing" may be split into "test", "#er", "#ed", and "#ing". They share the same sub-word "test" and also share similar meanings. However, for programming languages, a developer can use "test_str", "testStr" or "testString" to refer to variables of string type. The meanings of sub-words "test" and "str" are different, and the sub-word "str" may not refer to the word "string", as it can refer to any words starting with "str". Hence, a BERT model which is pre-trained on natural languages may be biased when it is directly applied for code embedding generation. However, ELMo, which can construct word embeddings from the character level, may not have the aforementioned issues. Therefore, as demonstrated by the results, ELMo could have better performance for vulnerable function detection on our data sets, compared to other mainstream word embedding models.

Table 3. The comparison between the ELMo module, Word2Vec, GloVe, FastText and BERT model in terms of their effectiveness of generating representations for source code functions on open source projects: FFmpeg, LibTIFF and OpenSSL. The generated representations are used as features to train a random forest (RF) classifier.

Software Project	**Embedding Model**	**# of Vulnerable Functions Found in Top-*k* (Top-*k* Precision)**							
		Top 10	**Top 20**	**Top 30**	**Top 40**	**Top 50**	**Top 100**	**Top 150**	**Top 200**
FFmpeg	ELMo + RF	9 (90%)	18 (90%)	22 (73%)	26 (65%)	31 (62%)	48 (48%)	56 (37%)	63 (32%)
	Word2Vec + RF	8 (80%)	14 (70%)	16 (53%)	20 (50%)	21 (42%)	32 (32%)	39 (26%)	49 (25%)
	FastText + RF	8 (80%)	13 (65%)	17 (57%)	19 (48%)	21 (42%)	32 (32%)	43 (29%)	51 (26%)
	GloVe + RF	7 (70%)	9 (45%)	12 (40%)	15 (38%)	17 (34%)	27 (27%)	37 (25%)	45 (23%)
	BERT + RF	7 (70%)	9 (45%)	11 (37%)	12 (30%)	18 (36%)	27 (27%)	38 (25%)	44 (22%)
LibTIFF	ELMo + RF	8 (80%)	13 (65%)	16 (53%)	20 (50%)	26 (52%)	42 (42%)	48 (32%)	52 (26%)
	Word2Vec + RF	5 (50%)	9 (45%)	11 (37%)	13 (33%)	16 (32%)	26 (26%)	38 (25%)	48 (24%)
	FastText + RF	8 (80%)	13 (65%)	15 (50%)	18 (45%)	20 (40%)	32 (32%)	41 (27%)	48 (24%)
	GloVe + RF	7 (70%)	12 (60%)	13 (43%)	17 (43%)	17 (34%)	30 (30%)	42 (28%)	52 (26%)
	BERT + RF	4 (40%)	9 (45%)	12 (40%)	13 (33%)	16 (32%)	30 (30%)	40 (27%)	51 (26%)
OpenSSL	ELMo + RF	9 (90%)	19 (95%)	27 (90%)	33 (83%)	38 (76%)	53 (53%)	60 (40%)	66 (33%)
	Word2Vec + RF	5 (50%)	9 (45%)	11 (37%)	13 (33%)	16 (32%)	26 (26%)	38 (25%)	48 (24%)
	FastText + RF	8 (80%)	15 (75%)	24 (80%)	27 (68%)	32 (64%)	38 (38%)	50 (33%)	53 (27%)
	GloVe + RF	8 (80%)	14 (70%)	19 (63%)	25 (63%)	26 (52%)	35 (35%)	39 (26%)	43 (22%)
	BERT + RF	8 (80%)	13 (65%)	17 (57%)	18 (45%)	22 (44%)	35 (35%)	40 (27%)	45 (23%)

3.4.2. The Structure of Bi-LSTM

To examine how different numbers of Bi-LSTM layers affect the learning of long-term dependency of the code sequences, we use different networks with different number of Bi-LSTM layers ranging from 0 to 3, and compare the results generated by these networks. As Figure 5 shows, when using only 1 Bi-LSTM layer in our framework, the best top-200 recall are achieved on all three projects. With the number of Bi-LSTM layer increases from 1 to 3, there is a continuous performance decrease as measured by the top-200 recall for all projects, except for LibTIFF. However, compared with using only 1 Bi-LSTM layer, a performance drop is still observed for LibTIFF. With more Bi-LSTM layers added to the network, the time required for training increases dramatically. Hence, adding more bi-LSTM layers does not contribute to performance increase for vulnerability detection on our data sets. For the following experiments, our framework contains only one Bi-LSTM layer.

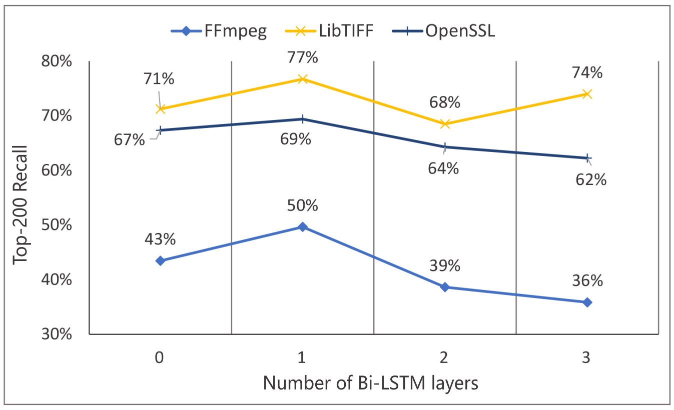


Figure 5. The top-200 recall of our framework with different number of Bi-LSTM layers on projects: FFmpeg, LibTIFF and OpenSSL.

3.4.3. Performance Evaluation and Comparison on Real-World Open Source Projects

Table 4 lists the results of performance comparison among our framework, Cross-VD [14], Multi-VD [27], DeepBalance [27] and Flawfinder, on open-source projects, FFmpeg, LibTIFF and OpenSSL. In general, the embeddings generated by our framework produce better results with RF and/or SVM than the other four systems on three projects. When using the embeddings generated by our framework, the results achieved by RF are better than those achieved by SVM in the majority of cases.

As Table 2 shows, the test set of project FFmpeg contains 143 vulnerable and 3820 non-vulnerable samples. Retrieving 10 of the most probable vulnerable functions could identify 9 actual vulnerable ones, using our framework. In comparison, using the method proposed by Cross-VD finds only 7 vulnerable functions with the same number of functions retrieved. The other two baseline systems could also identify 9 vulnerable functions. When returning 200 of the most likely vulnerable functions, our framework finds 72 vulnerable ones. Namely, using our method to examine 200 functions ranked by the probabilities of being vulnerable, one could identify 72 out of 143 total vulnerable functions from more than 3800 functions. This result outperforms the other four baseline systems. In contrast, Multi-VD [27] and DeepBalance[38] could find 70 and 71 vulnerable functions, respectively, which is very close to our method. Flawfinder returns 519 potentially vulnerable functions on the same test set. There are 44 actual vulnerable functions included. However, none of the 44 vulnerable functions are in the top 200 list based on the ranked severity level returned by the Flawfinder.

Despite project LibTIFF having the smallest number of function samples for training, using our method with SVM classifier achieves the best performance, compared with the performance achieved on the other two projects, which is 59 out of 64 vulnerable functions found when retrieving the top 200 ranked vulnerable functions. Using the other four systems, Cross-VD [14] could only find 49 vulnerable functions and the figures for Multi-VD [27], DeepBalance [38], and Flawfinder are 53, 56, and 12, respectively. On project OpenSSL, however, the performance deviation among four baseline detection systems is not as obvious as on the other two projects. Using our proposed framework, one could find 68 out of 96 vulnerable functions when retrieving 200 potentially vulnerable functions. In contrast, the performance of Multi-VD [27] and DeepBalance [38] is similar, with 65 and 67 vulnerable functions found, respectively.

Table 4. The results of performance comparison among our framework (with random forest (RF) and SVM), Cross-VD [14], Multi-VD [27], DeepBalance [38] and Flawfinder on three open-source projects: FFmpeg, LibTIFF, and OpenSSL.

Software Project	**Detection System**	**# of Vulnerable Functions Found in Top-*k* (Top-*k* Precision)**							
		Top 10	**Top 20**	**Top 30**	**Top 40**	**Top 50**	**Top 100**	**Top 150**	**Top 200**
FFmpeg	Our framework + RF	9 (90%)	18 (90%)	24 (80%)	31 (78%)	39 (78%)	61 (61%)	65 (43%)	72 (36%)
	Our framework + SVM	9 (90%)	16 (80%)	19 (63%)	23 (58%)	29 (58%)	51 (51%)	61 (41%)	65 (33%)
	Cross-VD [14]	7 (70%)	9 (45%)	14 (47%)	17 (43%)	18 (36%)	30 (30%)	39 (26%)	46 (23%)
	Multi-VD [27]	9 (90%)	14 (70%)	20 (67%)	24 (60%)	28 (56%)	47 (47%)	60 (40%)	70 (35%)
	DeepBalance[38]	9 (90%)	15 (75%)	20 (67%)	26 (65%)	33 (66%)	61 (61%)	63 (42%)	71 (36%)
	Flawfinder	Found 44 vulnerable functions, but not in the Top 200 list							
LibTIFF	Our framework + RF	7 (70%)	15 (75%)	22 (73%)	25 (63%)	28 (56%)	41 (41%)	50 (33%)	56 (28%)
	Our framework + SVM	8 (80%)	14 (70%)	18 (60%)	21 (53%)	27 (54%)	40 (40%)	50 (33%)	59 (30%)
	Cross-VD [14]	9 (90%)	14 (70%)	18 (60%)	19 (48%)	24 (48%)	34 (34%)	39 (26%)	49 (25%)
	Multi-VD [27]	8 (80%)	12 (60%)	20 (67%)	22 (55%)	26 (52%)	41 (41%)	46 (31%)	53 (27%)
	DeepBalance [38]	8 (80%)	15 (75%)	20 (67%)	21 (53%)	25 (50%)	41 (41%)	45 (30%)	56 (28%)
	Flawfinder	0	0	1 (3%)	2 (5%)	2 (4%)	7 (7%)	12 (8%)	12 (6%)
OpenSSL	Our framework + RF	9 (90%)	19 (95%)	28 (93%)	33 (83%)	40 (80%)	58 (58%)	64 (43%)	68 (34%)
	Our framework + SVM	9 (90%)	19 (95%)	28 (93%)	34 (85%)	41 (82%)	57 (57%)	63 (42%)	66 (33%)
	Cross-VD [14]	9 (90%)	19 (95%)	28 (93%)	31 (78%)	32 (64%)	43 (43%)	56 (37%)	61 (31%)
	Multi-VD [27]	8 (80%)	19 (95%)	28 (93%)	30 (75%)	39 (78%)	56 (56%)	63 (42%)	65 (33%)
	DeepBalance[38]	8 (80%)	19 (95%)	29 (97%)	33 (83%)	41 (82%)	57 (57%)	63 (42%)	67 (34%)
	Flawfinder	0	0	0	0	5 (10%)	55 (55%)	56 (37%)	56 (28%)

3.4.4. Detection of Context-Related Vulnerabilities

Table 5 presents the results of the number and the proportion of CWE-119 vulnerabilities accounted for all the found vulnerabilities in top-*k* on projects: FFmpeg, LibTIFF, and OpenSSL. As Table 5 shows, there are 47 CWE-119 vulnerabilities in the test set of FFmpeg, accounting for approximately 33% of the total vulnerabilities in the test set. When returning 10 of the most probable vulnerable functions, there are 9 actually vulnerable ones, and 4 of them are CWE-119 vulnerabilities, which account for 44%. When returning 20 of the most probable vulnerable functions, there are 18 actually vulnerable functions, and 10 of them are CWE-119 vulnerabilities, accounting for 56% of the total vulnerable functions. Similarly, when returning 30, 40, and 50 functions, the proportions of CWE-119 vulnerabilities are 50%, 48%, and 44%, respectively, which all exceed 33%. This indicates that our proposed framework is more likely to detect CWE-119 vulnerabilities that are context related.

For projects LibTIFF and OpenSSL, similar results are observed. As Table 5 shows, in most of the cases, our framework can detect more CWE-119 vulnerabilities, except for the case on LibTIFF where 20 functions are returned and 15 are actually vulnerable, among which 10 vulnerable functions belong to the CWE-119 category, accounting for 67% of the total. For other cases, our framework can detect more CWE-119 vulnerabilities than the other types, which is indicated by the proportions of the CWE-119 in the found vulnerabilities being higher than the proportion of the CWE-119 in all vulnerabilities in the test set.

Table 5. The number and the proportion of buffer errors (CWE-119) vulnerabilities accounted for in the found vulnerabilities in Top-*k* on projects: FFmpeg, LibTIFF, and OpenSSL.

Software Project	Our Framework	The Number and the Proportion of Buffer Errors Vulnerabilities in the Total Found Vulnerabilities in Top-*k*					The Number and the Proportion of Buffer Errors Vulnerabilities Accounted for All Vulnerabilities in the Test Set
		Top 10	Top 20	Top 30	Top 40	Top 50	
FFmpeg	with RF	4 (44%)	10 (56%)	12 (50%)	15 (48%)	17 (44%)	47 (33%)
	with SVM	4 (44%)	9 (56%)	10 (53%)	12 (52%)	13 (45%)	
LibTIFF	with RF	6 (86%)	10 (67%)	16 (73%)	18 (72%)	19 (68%)	43 (67%)
	with SVM	6 (75%)	10 (71%)	14 (78%)	17 (21%)	19 (70%)	
OpenSSL	with RF	2 (22%)	5 (26%)	6 (21%)	8 (24%)	9 (23%)	17 (18%)
	with SVM	2 (22%)	5 (26%)	6 (21%)	8 (24%)	9 (22%)	

3.5. Limitations and Future Work

There are several limitations of our proposed framework, which motivate further improvement and extensions. Firstly, in terms of measuring the effectiveness of generating code representations for vulnerable function detection, our work performs comparisons of ELMo with Word2vec, FastText, GloVe, and BERT, which are models originated from NLP fields, while neglecting the methods, such as Code2Vec [16] and code semantic representation generation [44], which are naturally built for generating code representations. Our future work will bridge this gap by including the recent code semantic representation generation techniques to explore whether these methods can produce more effective code representations with more semantic information preserved, eventually leading to improved vulnerability detection performance.

Secondly, when truncating the overly long sequences to a unified length of 1000, it is possible to truncate the actual vulnerable code parts which do not contribute to the first 1000 characters. One of the solutions is to extend the length threshold to accommodate more elements (i.e., code tokens) of the overly long sequences. However, the performance of the LSTM network degrades when the sequence length increases, due to several reasons (e.g., the hidden state bottleneck [45]). A trade-off is made between the length of the input sequences and the information loss caused by truncation. Considering that the vulnerable function sequences that are longer than 1000 only account for a small proportion, we believe that the vulnerable samples that have their actual vulnerable content removed during the truncation process would be very few in number and would not bias the classifier to a large extent. In addition, it may not be an ideal practice to have a function containing more than 1000 tokens.

Thirdly, our method is not able to provide interpretations for the detection results. On one hand, we cannot explain the reasons that the proposed network structure with ELMo, being a contextualized model, and Bi-LSTM, which can handle the long-range dependencies of sequences, facilitates the detection of certain type(s) of vulnerabilities (e.g., the taint-style vulnerabilities [33]). On the other, the neural network is used as a "black box", namely, the decision process of the network is unclear. Without the explanation of detection results, a code inspector may have to check the results manually. Thus, the inability to provide explanations for detection results can be one of the major barriers that hinder the wide adoption of neural models for vulnerability detection. Offering interpretations for neural model-based vulnerability detection methods can be one of our future research directions.

Lastly, more effective code embedding solutions and more expressive models are desirable for better capturing the complex and flexible vulnerable patterns. The proposed method builds on the structure of Bi-LSTM, which was proved to be inferior to many pre-trained language models in terms of code semantic understanding and contextual learning. Therefore, our future work may focus on utilizing CodeBERT [46] or CuBERT [47], which are contextualized models pre-trained on a large amount of code corpus for learning vulnerable features. In addition, we will continue to label more vulnerable functions based on open-source software projects, forming a real-world function-level vulnerability data set so that the robust and statistically stable neural models can be trained to contribute to improved detection performance.

4. Related Work

4.1. Embedding Methods for Source Code

The bimodal property of programming languages: understood by computers and readable by humans has motivated researchers to apply NLP techniques, including neural language models to software code analysis [16,17,48]. Different from conventional ML techniques, which primarily rely on labor-intensive feature engineering, neural network models are capable of learning latent patterns which can be more generalizable to the task of interest. However, there is a “domain gap" between the neural network, which was originally designed to cope with the numeric data and software code, which consists of textual tokens being “discrete" in nature. To bridge the gap, various code encoding methods were used. One-hot encoding was adopted by many studies which applied neural techniques to code analysis. White et al. [19] used one-hot encoding for representing source code tokens. One-hot encoding was also used at the character level [20,21] .

Unfortunately, one-hot encoded vectors fail to capture the similarities/differences of the items they represent. Hence, different methods were proposed to generate code embeddings as distributed representations. In the field of NLP, the distributed representations of words (e.g., Word2Vec) capture the meanings of words distributed in the components of vectors [16]. With Word2Vec model, words can be encoded, using meaningful vector representations to better facilitate the learning tasks. For this reason, many existing studies that applied neural models for code analysis used Word2Vec model for learning code embeddings (see, for example, [14,25–27,49–51]).

Embedding techniques, such as Word2Vec, failed to generate different representations for the same word used in different contexts. To better optimize the learning of contextual representation of words, methods such as Context2Vec [29] and ELMo [15] were proposed. To transfer source code to vector representations, [16,52,53] applied deep learning techniques to generate code embeddings based on ASTs. Our work differs with the aforementioned studies in two main points: firstly, our trained framework takes directly source code as inputs, which does not require any code analysis tools for further processing and outputs the learned function-level embeddings. Secondly, our framework applies ELMo, which is a contextualized model, for generating code representations based on contexts. Additionally, the Bi-LSTM structure also helps to capture the long-range dependencies of sequences. Due to vulnerability detection requiring the understanding of code contexts to better track the data flows and/or control flows, our proposed Bi-LSTM framework incorporating ELMo can better identify vulnerable patterns related to contexts.

The aforementioned embedding methods originate from the field of NLP. There is a line of studies which extract the code embeddings from the syntactic tree representations and intermediate representations (IRs) of software code. Alon et al. [16] proposed Code2Vec, which decomposes a source code function to a collection of paths in its AST, allowing a neural network to learn the representation of each path and use the attention mechanism to select the relevant paths and aggregate a set of them to form a fixed-length vector. Later, the authors proposed code2seq [54], which is an improved method for representing source code with the syntactic structure of programming languages. However, both Code2Vec and Code2Seq are programming language dependent. Ben-Nun et al. [44] proposed inst2vec, which produces code representations based on IRs, which are based on the data flow and control flow of a program, making the generated representations programming language independent.

To facilitate reverse engineering, Asm2Vec was proposed to generate robust representations for assembly code, which can prevent changes brought by obfuscation and optimizations [55]. Similarly, Zuo et al. proposed a neural machine translation (NMT) based framework, which can convert basic blocks of assembly languages to semantically meaningful representations [56]. Both methods can be applied for detecting binary-level vulnerabilities and provide insights and novel ideas for developing more effective code embedding solutions in future work.

4.2. Software Vulnerability Detection

Rules derived from the experience of knowledgeable individuals are applied for detecting potential buggy/vulnerable code that did not conform to the rules or best practices [37,57,58]. However, designing rules to cover all possible programming flaws is infeasible. The ML techniques offer new solutions for bugs/vulnerabilities detection by learning rules/patterns automatically. For instance, Neuhaus et al. [59] applied features from imports and function calls as indicators of vulnerable software components. API usage patterns were used by Yamaguchi et al. [60] as features for predicting functions containing potential vulnerabilities. Perl et al. [6] extracted code-based metrics and meta-data retrieved from open-source projects to predict commits that lead to vulnerabilities. Nevertheless, the aforementioned studies still depend on the manual process for engineering features, which can be time consuming and also error prone.

Recently, deep learning techniques were applied for bug/vulnerability detection. It is hypothesized that the deep architecture is capable of automatically learning the latent patterns, which can be more effective and generalizable than the features driven by human knowledge. Based on this hypothesis, Wang et al. [13] used deep belief networks for learning semantic patterns for bug discovery. Lin et al. [14], and Lin et al. [49] leveraged LSTM network for learning high-level representations for detecting vulnerabilities on C software projects. Li et al. [61] adopted LSTM network to learn vulnerable patterns on so-called “code gadgets”, which are the explicitly-defined data flows. The data flow can either be within function boundary or span multiple functions. In this paper, we combine the ELMo module and LSTM network to learn latent source-sink patterns directly from the source code, and thus no other code analysis tools are required. In terms of learning source-sink patterns that are associated with taint-style vulnerabilities, Yamaguchi et al. [33] extended the code property graph (CPG) [62] for analyzing the vulnerable source-sink patterns. They converted source code to graphs and formulated the vulnerable source-sink patterns as graph traversal queries for matching potential vulnerabilities. Our work utilizes the deep neural network for learning vulnerable source-sink patterns automatically, thus significantly simplifying the code analysis processes.

5. Conclusions

In this paper, we propose a supervised framework to extract code embeddings for function-level vulnerability detection. The framework is designed to capture the source-sink data flow that does not involve any validation. Specifically, our framework builds on the ELMo model to allow the contextual semantics hidden in the source code to be learned. By doing this, the textual code sequences are converted to meaningful dense vectors and fed to the Bi-LSTM layer for further learning of long-range dependency. This enables the network to capture the potentially vulnerable code sequences. To allow the extracted code embeddings to be acceptable for mainstream ML classifiers, we use a global max pooling layer to convert the learned embeddings as vectors. In addition, our framework takes the source code as the input without the need for code analysis. The experimental study showed that the code embeddings generated by our framework are effective feature sets for vulnerability detection. With the embeddings produced by our framework, the detection results achieved by random forest outperformed the four baseline systems on our real-world software projects.

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

Funding: This work was supported in part by the Natural Science Foundation Project of Fujian Province, China, under Grant 2021J011131 and 2021J011128, and in part by the National innovation and Entrepreneurship Program for College Students, under Grant 202111311014.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: A part of the experimental data and code for reproducing this work can be found at Github: https://github.com/danielLin1986/Function-level-Vulnerability-Detection, accessed on 10 November 2021.

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

References

1. Sun, N.; Zhang, J.; Rimba, P.; Gao, S.; Zhang, L.Y.; Xiang, Y. Data-driven cybersecurity incident prediction: A survey. *IEEE Commun. Surv. Tutorials* **2019**, *21*, 1744–1772. [CrossRef]
2. Liu, L.; De Vel, O.; Han, Q.L.; Zhang, J.; Xiang, Y. Detecting and Preventing Cyber Insider Threats: A Survey. *IEEE Commun. Surv. Tutorials* **2018**, *20*, 1397–1417. [CrossRef]
3. Chen, X.; Li, C.; Wang, D.; Wen, S.; Zhang, J.; Nepal, S.; Xiang, Y.; Ren, K. Android HIV: A study of repackaging malware for evading machine-learning detection. *IEEE Trans. Inf. Forensics Secur.* **2020**, *15*, 987–1001. [CrossRef]
4. Coulter, R.; Han, Q.L.; Pan, L.; Zhang, J.; Xiang, Y. Data-driven cyber security in perspective–intelligent traffic analysis. *IEEE Trans. Cybern.* **2020**, *50*, 3081–3093. [CrossRef] [PubMed]
5. 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. *arXiv* **2018**, arXiv:1801.01681.
6. Perl, H.; Dechand, S.; Smith, M.; Arp, D.; Yamaguchi, F.; Rieck, K.; Fahl, S.; Acar, Y. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, Denver, CO, USA, 12–16 October 2015; pp. 426–437.
7. Qiu, J.; Zhang, J.; Luo, W.; Pan, L.; Nepal, S.; Xiang, Y. A survey of Android malware detection with deep neural models. *ACM Comput. Surv.* **2020**, *53*, 1–36. [CrossRef]
8. Wang, M.; Zhu, T.; Zhang, T.; Zhang, J.; Yu, S.; Zhou, W. Security and Privacy in 6G Networks: New Areas and New Challenges. *Digit. Commun. Netw.* **2020**, *6*, 281–291. [CrossRef]
9. 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]
10. Lin, G.; Wen, S.; Han, Q.; Zhang, J.; Xiang, Y. Software Vulnerability Detection Using Deep Neural Networks: A Survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
11. Zeng, P.; Lin, G.; Pan, L.; Tai, Y.; Zhang, J. Software Vulnerability Analysis and Discovery using Deep Learning Techniques: A Survey. *IEEE Access* **2020**. [CrossRef]
12. Chen, Z.; Monperrus, M. A literature study of embeddings on source code. *arXiv* **2019**, arXiv:1904.03061.
13. 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; pp. 297–308.
14. 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]
15. Peters, M.E.; Neumann, M.; Iyyer, M.; Gardner, M.; Clark, C.; Lee, K.; Zettlemoyer, L. Deep contextualized word representations. *arXiv* **2018**, arXiv:1802.05365
16. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning Distributed Representations of Code. *arXiv* **2018**, arXiv:1803.09473.
17. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* **2018**, *51*, 81. [CrossRef]
18. Scandariato, R.; Walden, J.; Hovsepyan, A.; Joosen, W. Predicting vulnerable software components via text mining. *TSE* **2014**, *40*, 993–1006. [CrossRef]
19. White, M.; Vendome, C.; Linares-Vásquez, M.; Poshyvanyk, D. Toward deep learning software repositories. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015.
20. Karpathy, A.; Johnson, J.; Fei-Fei, L. Visualizing and understanding recurrent networks. *arXiv* **2015**, arXiv:1506.02078.
21. Cummins, C.; Petoumenos, P.; Wang, Z.; Leather, H. Synthesizing benchmarks for predictive modeling. In Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Austin, TX, USA, 4–8 February 2017.
22. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed Representations of Words and Phrases and Their Compositionality. Available online: https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf (accessed on 10 November 2021).

23. Pennington, J.; Socher, R.; Manning, C. Glove: Global Vectors for Word Representation. Available online: https://aclanthology.org/D14-1162.pdf (accessed on 10 November 2021).
24. Bojanowski, P.; Grave, E.; Joulin, A.; Mikolov, T. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguist.* **2017**, *5*, 135–146. [CrossRef]
25. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z.; Wang, S.; Wang, J. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *arXiv* **2018**, arXiv:1807.06756.
26. Harer, J.A.; Kim, L.Y.; Russell, R.L.; Ozdemir, O.; Kosta, L.R.; Rangamani, A.; Hamilton, L.H.; Centeno, G.I.; Key, J.R.; Ellingwood, P.M.; et al. Automated software vulnerability detection with machine learning. *arXiv* **2018**, arXiv:1803.04497.
27. 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 Secur. Comput.* **2019**. [CrossRef]
28. Lin, G.; Xiao, W.; Zhang, L.Y.; Gao, S.; Tai, Y.; Zhang, J. Deep neural-based vulnerability discovery demystified: data, model and performance. *Neural Comput. Appl.* **2021**, *33*, 1–14. [CrossRef]
29. Melamud, O.; Goldberger, J.; Dagan, I. context2vec: Learning Generic Context Embedding with Bidirectional Lstm. Available online: https://aclanthology.org/K16-1006.pdf (accessed on 10 November 2021).
30. Gardner, M.; Grus, J.; Neumann, M.; Tafjord, O.; Dasigi, P.; Liu, N.H.; Peters, M.; Schmitz, M.; Zettlemoyer, L.S. A Deep Semantic Natural Language Processing Platform. *arXiv* **2017**, arXiv:1803.07640.
31. Chelba, C.; Mikolov, T.; Schuster, M.; Ge, Q.; Brants, T.; Koehn, P.; Robinson, T. One billion word benchmark for measuring progress in statistical language modeling. *arXiv* **2013**, arXiv:1312.3005.
32. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv* **2018**, arXiv:1810.04805.
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. Keskar, N.S.; Mudigere, D.; Nocedal, J.; Smelyanskiy, M.; Tang, P.T.P. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv* **2016**, arXiv:1609.04836.
35. National Vulnerability Database. Available online: https://nvd.nist.gov/ (accessed on 28 September 2021).
36. Common Vulnerabilities and Exposures. Available online: https://cve.mitre.org/index.html (accessed on 26 September 2021).
37. Wheeler, D.A. Flawfinder. 2021. Available online: https://www.dwheeler.com/flawfinder/ (accessed on 20 September 2021).
38. Liu, S.; Lin, G.; Han, Q.L.; Wen, S.; Zhang, J.; Xiang, Y. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Trans. Fuzzy Syst.* **2019**, *28*, 1329–1343. [CrossRef]
39. Keras-team. Keras. 2015 Available online: https://github.com/fchollet/keras (accessed on 10 November 2021).
40. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A System for Large-Scale Machine Learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), Savannah, GA, USA, 2–4 November 2016.
41. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *JMLR* **2011**, *12*, 2825–2830.
42. Řehůřek, R.; Sojka, P. Software Framework for Topic Modelling with Large Corpora. In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, Valletta, Malta, 22 May 2010; pp. 45–50. Available online: http://is.muni.cz/publication/884893/en (accessed on 10 November 2021).
43. Christopher, D.; Manning, P.R.; Schütze, H. *Introduction to Information Retrieval*; Cambridge University Press: Cambridge, UK, 2009; Chapter 8, pp. 151–175.
44. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *arXiv* **2018**, arXiv:1806.07336.
45. Bhoopchand, A.; Rocktäschel, T.; Barr, E.; Riedel, S. Learning Python code suggestion with a sparse pointer network. *arXiv* **2016**, arXiv:1611.08307.
46. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
47. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and evaluating contextual embedding of source code. In Proceedings of the International Conference on Machine Learning, PMLR, Vienna, Austria, 13–18 July 2020.
48. Hindle, A.; Barr, E.T.; Su, Z.; Gabel, M.; Devanbu, P. On the naturalness of software. In Proceedings of the 34th International Conference on Software Engineering (ICSE), New Delhi, India, 21–23 February 2013.
49. Lin, G.; Zhang, J.; Luo, W.; Pan, L.; Xiang, Y. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In Proceedings of the CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017.
50. Lin, G.; Xiao, W.; Zhang, J.; Xiang, Y. Deep learning-based vulnerable function detection: A benchmark. In Proceedings of the International Conference on Information and Communications Security, Beijing, China, 15–17 December 2019.
51. Jia, H.; Peng, X.; Lang, C. Remora optimization algorithm. *Expert Syst. Appl.* **2021**, *185*, 115665. [CrossRef]

52. Peng, H.; Mou, L.; Li, G.; Liu, Y.; Zhang, L.; Jin, Z. Building program vector representations for deep learning. In Proceedings of the International Conference on Knowledge Science, Engineering and Management, Chongqing, China, 28–30 October 2015.
53. 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.
54. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv* **2018**, arXiv:1808.01400.
55. Ding, S.H.; Fung, B.C.; Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019.
56. Zuo, F.; Li, X.; Young, P.; Luo, L.; Zeng, Q.; Zhang, Z. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv* **2018**, arXiv:1808.04706.
57. Rough-Auditing-Tool-for-Security. Available online: https://code.google.com/archive/p/rough-auditing-tool-for-security/ (accessed on 26 September 2021).
58. Engler, D.; Chen, D.Y.; Hallem, S.; Chou, A.; Chelf, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SIGOPS Operating Systems Review*; ACM: New York, NY, USA, 2001; Volume 35, pp. 57–72.
59. Neuhaus, S.; Zimmermann, T.; Holler, C.; Zeller, A. Predicting vulnerable software components. In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 29 October–2 November 2007.
60. Yamaguchi, F.; Lindner, F.; Rieck, K. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In Proceedings of the 5th USENIX Workshop on Offensive Technologies, San Francisco, CA, USA, 8 August 2011.
61. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Qi, H.; Hu, J. VulPecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, Los Angeles, CA, USA, 5–8 December 2016.
62. 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, Berkeley, CA, USA, 18–21 May 2014.