Cross-Project Transfer Representation Learning for Vulnerable Function Discovery

Guanjun Lin, Jun Zhang, *Member, IEEE*, Wei Luo, Lei Pan, *Member, IEEE*, Yang Xiang, *Senior Member, IEEE*, Olivier De Vel, and Paul Montague

***Abstract*—Machine learning is now widely used to detect security vulnerabilities in the software, even before the software is released. But its potential is often severely compromised at the early stage of a software project when we face a shortage of high-quality training data and have to rely on overly generic hand-crafted features. This paper addresses this cold-start problem of machine learning, by learning rich features that generalize across similar projects. To reach an optimal balance between feature-richness and generalizability, we devise a data-driven method including the following innovative ideas. First, the code semantics are revealed through serialized abstract syntax trees (ASTs), with tokens encoded by Continuous Bag-of-Words neural embeddings. Next, the serialized ASTs are fed to a sequential deep learning classifier (Bi-LSTM) to obtain a representation indicative of software vulnerability. Finally, the neural representation obtained from existing software projects is then transferred to the new project to enable early vulnerability detection even with a small set of training labels. To validate this vulnerability detection approach, we manually labeled 457 vulnerable functions and collected 30 000+ nonvulnerable functions from six open-source projects. The empirical results confirmed that the trained model is capable of generating representations that are indicative of program vulnerability and is adaptable across multiple projects. Compared with the traditional code metrics, our transfer-learned representations are more effective for predicting vulnerable functions, both within a project and across multiple projects.**

***Index Terms*—Abstract syntax tree, cross-project, representation learning, transfer learning, vulnerability discovery.**

Manuscript received March 21, 2018; accepted March 26, 2018. Date of publication April 2, 2018; date of current version July 2, 2018. Paper no. TII-18-0714. *(Corresponding author: Jun Zhang.)*

G. Lin, W. Luo, L. Pan are with the School of Information Technology, Deakin University, Geelong, VIC 3216, Australia (e-mail: lingu@deakin.edu.au; wei.luo@deakin.edu.au; l.pan@deakin.edu.au).

J. Zhang is with the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: junzhang@swin.edu.au).

Y. Xiang is with the Digital Research & Innovation Capability Platform, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: yxiang@swin.edu.au).

O. De Vel and P. Montague are with the Defence Science & Technology Group, Department of Defence, Maribyrnong, VIC 3032, Australia (e-mail: Olivier.DeVel@dst.defence.gov.au; Paul.Montague@dst.defence.gov.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2018.2821768

I. Introduction

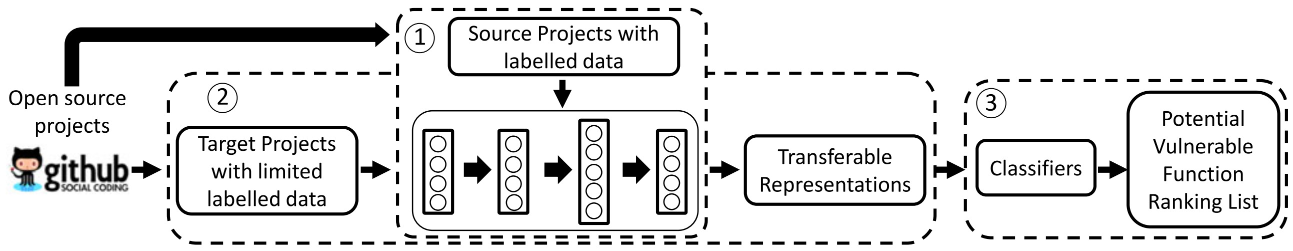
VULNERABILITIES in software critically undermine the security of computer systems and threaten the IT infrastructure of many government sectors and organizations. For instance, the recently disclosed "*Heartbleed*" and "*Shellshock*" vulnerabilities, and a vulnerability in the server message block (SMB) protocol exploited by the WannaCry ransomware have affected a wide range of systems and millions of users worldwide. According to [4] and [26], one of the major causes of security incidents and breaches can be attributed to the exploitable vulnerabilities in software. Once a vulnerability is exploited by attackers, companies and organizations may suffer from significant financial loss as well as irreparable damage to their reputation [22].

The early detection of vulnerabilities in applications is vital for implementing cost-effective attack-mitigation solutions. From the perspective of code execution, techniques for identifying vulnerabilities can be categorized into static, dynamic, and hybrid approaches. Static techniques, such as rule-based analysis [6], code similarity detection i.e., code clone detection [8], [9], and symbolic execution [2], mainly rely on the analysis of source code, but often struggle to reveal bugs and vulnerabilities occurring at the runtime. Dynamic analysis includes fuzzing test [23] and taint analysis [17], and focuses on detecting vulnerabilities manifested during program execution, but in general, has low-code coverage. The hybrid approaches combining static and dynamic analysis techniques aim to overcome the aforementioned weaknesses. However, all of these approaches rely on a limited set of known syntactic or behavioral patterns of vulnerabilities, and such deficiency raises the challenge of detecting the previously unseen vulnerabilities.

Data-driven vulnerability discovery using machine learning (ML) provides a new opportunity for intelligent, effective, and efficient vulnerability detection. The existing ML-based approaches primarily operate on source code, which offers better human readability. Researchers have applied source-code based features, such as imports (i.e., header files), function calls [16], software complexity metrics, and code changes [22], as indicators for identifying potentially vulnerable files or code fragments. Moreover, features and information obtained from version control systems, such as developer activities [12] and code commits [20], were also adopted for predicting vulnerabilities. Most recently, two studies: 1) VUDDY [9]; and 2) VulPecker [10], focused on detecting vulnerable

1551-3203 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.



Open source projects
github SOCIAL CODING
1
Source Projects with labelled data
2
Target Projects with limited labelled data
Transferable Representations
3
Classifiers
Potential Vulnerable Function Ranking List

Fig. 1. Proposed framework for vulnerability discovery. It contains three stages: The first stage is to pretrain a bidirectional long short-term memory (Bi-LSTM) network using source code projects; the second stage is to feed the trained network with the target project to obtain representations as features; the last stage is to train an ML classifier with the learned features.

functions and code fragments based on code clone/similarity analysis, nevertheless, both approaches incur high false negative rate.

However, most of the existing ML-based approaches focus on software component- or file-level vulnerability detection, which rely on the manual effort and expertise of the code auditor to inspect the code base to accurately pinpoint the exact location of the vulnerabilities. Because of the relative scarcity of vulnerabilities, there is insufficient historical vulnerability data for training and validating a statistical model, especially on inactive open-source projects. In this paper, we aim to explore a fine-grained vulnerability detection approach targeting multiple software projects. To overcome the challenges, we propose a framework that solves this problem in three stages (see Fig. 1). First, we create vulnerability ground truth data at the function-level. Second, we extract features from the abstract syntax trees (ASTs) of each function. Specifically, we use a parser to obtain ASTs in a serialized form by using depth-first traversal (DFT). Then, we convert the serialized ASTs to equal-length sequences while preserving the structural and semantic features. To further refine these features, we apply a long short-term memory (LSTM) [7] recurrent neural network with Word2vec [13] embeddings for learning a higher level of representations. We hypothesize that the algorithm has the capacity of automatically extracting deep vulnerable programming features that contain richer information than the shallow features driven by domain knowledge. We also hypothesize that the learned low-level representations are transferable and are independent of software projects using the same programming language. Last, given a target project with insufficient labeled data, we apply the same feature-extraction process and feed the data to the pretrained network for learning a subset of representations. Subsequently, the learned representations are used to train a classifier for vulnerability prediction. The empirical study shows that the features extracted using our method are significantly more effective than software code metrics (CMs) in detecting vulnerabilities. Despite the small number of instances labeled in the projects, our algorithm is capable of effectively utilizing available data from other projects for pretraining a basic network, which can then be used for extracting deep AST representations for the projects with insufficient data. Empirical results demonstrate the effectiveness of learned representations which contribute to better detection accuracy than using traditional CMs.

In summary, our contributions are three-fold.

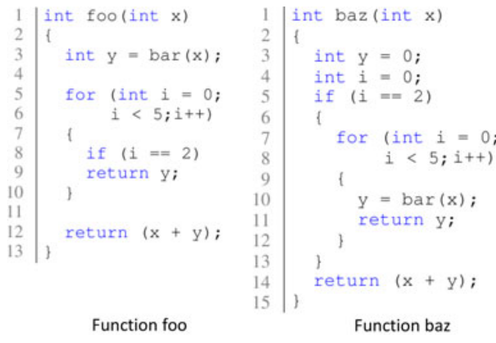
1) We propose a framework for function-level vulnerability discovery, which offers a fine-grained detection capability, facilitating a quick location of vulnerabilities.
2) We develop an approach to extract the sequential features of ASTs that capture the structural and semantic information of functions. Such information reflects the vulnerable programming patterns.
3) We construct a Bi-LSTM network for effectively extracting deep AST representations, which supports the transferability across software projects. The empirical studies show that the deep AST representations provide the precise identification of vulnerable functions (80% precision is achieved when retrieving the 10 most probable vulnerable functions).

The rest of this paper is organized as follows: Section II presents how features are extracted from ASTs derived from source code functions. Section III describes how to leverage LSTM for obtaining the sequential patterns in ASTs for vulnerability detection. Then, Section IV evaluates the detection performance of the proposed approach using two sets of experiments for the evaluation of the effectiveness of our deep AST representations and transfer-representation learning. Section VI concludes this paper.

II. Function Level AST Engineering

We believe that software vulnerabilities are often reflected in the syntactical structure of source code, particularly at the function-level. To capture such features and code properties, we follow the early work of Yamaguchi *et al.* [25]. The authors assumed that vulnerable programming patterns are associated with many vulnerabilities, and these patterns can be revealed by analyzing the program's ASTs. An AST is a syntactical structure of source code (for instance, a function), depicting the relationships among the components of the code in a hierarchical tree view, and faithfully representing the function-level control flow [see Fig. 2(a) and (b)]. Compared with the control flow graphs (CFGs), ASTs provide a natural program representation at the function level and reserve more information of the source code, while CFGs usually do not include variable declarations. Therefore, in this paper, we choose ASTs for extracting the latent programming patterns. To achieve this, an AST needs to be serialized for converting to a vector while preserving its

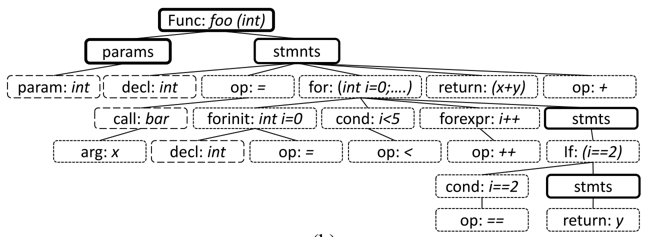
Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.



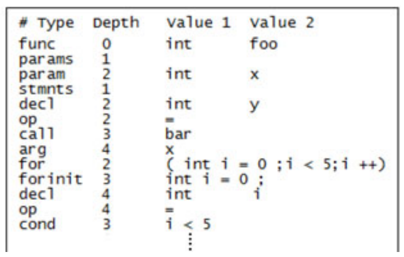
```
int foo(int x)
{
  int y = bar(x);

  for (int i = 0;
        i < 5;i++)
  {
    if (i == 2)
    return y;
  }

  return (x + y);
}
```

```
int baz(int x)
{
  int y = 0;
  int i = 0;
  if (i == 2)
  {
    for (int i = 0;
          i < 5;i++)
    {
      y = bar(x);
      return y;
    }
  }
  return (x + y);
}
```

(a)



(b)



```
# Type   Depth   Value 1   Value 2
func      0      int       foo
params    1
param     2      int       x
stmnts    1
decl      2      int       y
op        2      =
call      3      bar
arg       4      x
for       2      ( int i = 0 ;i < 5;i ++)
forinit   3      int i = 0 ;
decl      4      int        i
op        4      =
cond      3      i < 5
                    ⋮
```

(c)

Fig. 2. Motivating examples of how function *foo* is converted to an AST in serialized format. (a) Two examples of C functions. (b) An AST of function *foo*, with placeholder nodes (thick line), API nodes (dashed line) and syntax node (dotted line) [25]. (c) A serialized AST of function *foo*.

structure and semantics. The vector that holds the structural and semantic information can then be leveraged by our proposed Bi-LSTM network for obtaining deep representations capable of distinguishing vulnerable functions from nonvulnerable ones.

A. Robust AST Parsing

Prior to extracting features from ASTs, we have to obtain the ASTs from the source code. ASTs are usually generated by a compiler during the code parsing stage. However, without a working build environment, obtaining ASTs from C/C++ code is nontrivial. With "*CodeSensor*," which is a robust parser implemented by [25] based on the concept of *island grammars* [15], we can extract ASTs from individual source files or even from fragments of function code without the presence of dependent libraries. By feeding *CodeSensor* with the source code files, the parsed ASTs can be generated in a serialized format, ready for subsequent processing.

In Fig. 2(c), a serialized AST is organized to fit in a table, presenting a more user-friendly view than the original tree view. The first column is the type of all the nodes; the second column records the depth/layer of each type; and the last two columns present the actual names and values of the types in the original function, respectively.

When referring to the nodes of ASTs, we follow [25] for the naming conventions. An AST, as shown in Fig. 2(b), consists of three types of nodes: 1) placeholder nodes; 2) API nodes; and 3) syntax nodes. The placeholder nodes are not actual components of a function but they link the function components together to form a tree. All the ASTs have placeholder nodes, such as "*params*," signifying that its leaf nodes are function parameters, or "*stmnts*" denoting its leaf being statements of various types. According to [25], the API nodes refer to the types of function return values and function parameters. They also can be variable declarations and function calls. For instance, a function which has a "*void*" return type will have a "*void*" node. A function taking an *int* parameter will have an "*int*" node. The syntax nodes are syntactic elements that include control flow elements and operators. The control flow elements are derived from *while*/*do while* statements, *if*/*else* statements, etc. If a function contains an *if*/*else* statement, it will have "if" and "else" nodes in its AST. As for operators, such as "+," "−," or "=," they will remain unchanged.

B. AST Refining

1) Code Structure Preserving:

With the parsed ASTs in a serialized format, they need to be transformed to vectors so that they can be processed by ML algorithms. To preserve the structural information of ASTs, we applied the same method addressed in our previous work [11]. First, the ASTs have to be traversed, allowing their components to be assembled in a uniform sequence to form vectors. In this paper, we use a DFT to map the AST elements to vectors. In future work, we will examine whether breadth-first traversal yields better results in the subsequent classification process.

With serialized ASTs, using DFT is straightforward, as the serialized format [see Fig. 2(c)] is written in a depth-first search sequence from top to bottom. For every AST, we map its nodes to a vector so that each node becomes an element in the vector. To preserve the structural information of ASTs, the sequence of each element matters. Take function *foo* shown in Fig. 2(a) for example, the root of its AST [see Fig. 2(b)], being a function's name: *foo*, will be mapped to the first element of the vector. The second and third layer of the AST, which usually contain the "*params*" node, return type and the parameter type, will be the second, third, and fourth element of the vector respectively, and likewise for the subsequent nodes. After mapping the AST of function *foo* to a vector, it will be of the form like [*foo, int, params, int, x, stmnts, decl, int, y*, =, *call, bar*, ...]. For this textual vector, we treat it as a "sentence" with semantic meanings. The semantic meaning is formed by the elements of the vector and their sequence. For instance, function *baz* in Fig. 2(a) is similar to function *foo*, both of which take a parameter *x* of type *int* and return an *int* value. They also share the same names of local variables and have *for*, *if* statements and call the same function. However, they are different in terms of behavior. When mapping function *baz* to a vector: [*baz, int, params, int, x, stmnts, decl, int, y*, =, *decl, int*, ...], we can immediately recognize that the sequence of elements of vector *baz* differs from that of vector

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.

foo, in spite of them sharing the same textual content. Therefore, the converted vector should uniquely identify a function. By doing so, we can preserve the structural and contextual information to a large extent.

2) Tokenization and Padding: Since the subsequent ML algorithms take numeric vectors as inputs, textual elements of vectors are mapped to numbers. To achieve this, a mapping is built to link each textual element of vectors to an integer. These integers act as "tokens", which uniquely identify each textual element. For instance, we map type "int" to "1" and keyword "static" to "2," and so on. By replacing the textual elements of vectors with numeric tokens, their sequence remains intact.

Padding and truncation are standard practices to handle vectors of various length and provide a unified length of input vectors. Among the vectors resulting from the previous step, our longest vector contains more than 2200 elements and the shortest one is less than 5. To balance the length and the over-sparsity of vectors, a suitable length should be chosen for padding/truncation. For short vectors, we use zeros for end-of-sequence padding.

3) Code Semantics Preserving: Besides the structural information, the code semantics also need to be preserved through word embedding. The word embedding represents each "word" of the input sequence as a dense vector of a fixed dimension. The traditional embedding methods may convert words, in our context, the AST nodes, such as type "*int*," or operators like "+" and "−," to arbitrary dimensional vectors while ignoring the relationships that may exist between the nodes. To allow the algorithm to leverage the information that is held between the nodes (namely, the elements of vectors) and be more expressive, we apply Word2vec [13] with the continuous Bag-of-Words model to convert each element of the vectors to word embeddings of 100 dimensions (which is the default settings). We use the Word2vec directly at the output of *CodeSensor* and include all of the code base and the type of AST nodes as the corpus of text content for the algorithm to learn from. We visualize the embeddings learned using Word2vec and project the embeddings with principle component analysis (PCA) to a two-dimensional (2-D) plane. Fig. 3 shows different types of elements are grouped into separable clusters. Therefore, with Word2Vec, the elements of vectors can be represented with semantically meaningful vector representations so that the elements that share similar contexts in the code are located in close proximity in the vector space.

III. Transferable Representation Learning

Traditionally, the tree structure, such as an AST can be processed as shallow graph theoretical features. Such features are still not sufficiently robust across different software projects. In this paper, we leverage the recent advances in recurrent neural networks (RNNs) [14] to reveal deep sequential features that support transferability across different software projects. Based on RNNs, we proposed a Bi-LSTM network, using the LSTM cell as the building block, plus a global max pooling layer to accommodate large ASTs for extracting the latent sequential features in ASTs.

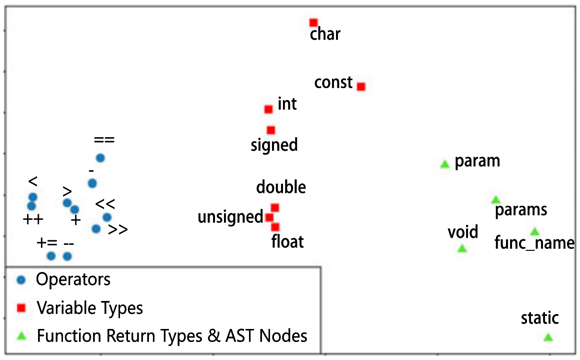


Fig. 3. Plot of AST node embeddings learned through Word2vec. PCA was used to project 100-dimensional vectors learned with Word2vec to a 2-D plane. The blue dots represent operators in ASTs; the red dots denote variable types; and the green ones are function return types and placeholder nodes. The figure depicts that AST nodes with similar code semantics (e.g., the operators) are grouped together, showing that applying Word2vec produces more meaningful embeddings.

A. RNN and LSTM

RNNs are capable of extracting complex sequential interactions that are crucial for some prediction tasks. They treat inputs as sequentially dependent data. For a given time step t, an RNN's output y_t not only depends on the current input x_t but also on the accumulated information up to the time step $t-1$. This feature offers RNNs the capability of retaining useful information from the past for the current prediction [18] so that they become powerful tools for solving Natural language processing (NLP) problems. There is a strong resemblance between ASTs and "*sentences*" in natural languages, which motivates us to apply RNNs to our scenario. Since, the vectors containing components of ASTs in a sequence reflect the structural information, altering the sequence of any element changes the structural and semantic meanings. For example, vector [*main, int, decl, int,* =, ..., *return*] means that the *main* function returning an *int* type and contains a declaration of a variable of *int* type. If we change the sequence of "*main*" and "*decl*" so that the vector [*decl, int, main, int, op,* ..., *return*] becomes semantically meaningless because a local variable declaration should be inside of the main function for C/C++. So, we hypothesize that a function with vulnerability will display certain "linguistic" oddities discoverable by RNNs.

Usually, the vulnerable programming patterns in a function can be associated with many lines of code. When we map functions to vectors, patterns linked to vulnerabilities are related to multiple elements of the vector. The standard RNNs are able to handle short-term dependencies, such as the element "*main*," which should be followed by type name "*int*" or "*void*," but they have a problem when dealing with long-term dependencies, such as capturing vulnerable programming patterns that are related to many continuous or intermittent elements. Therefore, we use an RNN with LSTM cells to capture long-term dependencies [18] for learning high-level representations of vulnerabilities.

B. Network Architecture

The architecture of our LSTM-based network is illustrated in Fig. 4. The configuration of the network (i.e., the choice of

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.

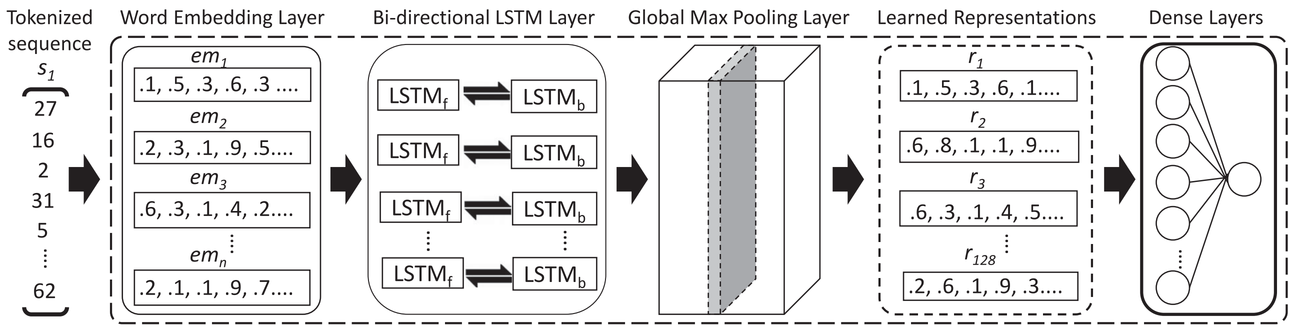


Fig. 4. Five-layer architecture of the proposed LSTM network for learning deep AST representations. During the pretraining phase, the network takes a tokenized sequence converted from an AST as an input. In the representation-learning phase, the last two dense layers are removed and the output of global max pooling layers are used as the learned deep AST representations as features for subsequent processing.

activation function, the number of LSTM cells) is fine-tuned, based on the experiments. The network takes a "tokens" sequence as input. The first layer is the Word2vec embedding layer mentioned in Section II-B3, which maps each element of the sequence to a vector in a semantic space where similar elements are close to each other. The second layer is an LSTM layer which contains 64 LSTM units in a bidirectional form (a total of 128 LSTM units). The third layer is a global max pooling layer. Since in our datasets, each function sample contains at most one vulnerability, applying global max pooling to select the maximum value can help to strengthen potentially vulnerable signals. During the training phase, two dense layers, one of which with the sigmoid activation function, are added for converging the learned representations to a probability, indicating how likely this sequence is vulnerable. We acquire the output (activations) of the third layer as the learned representative features that are highly abstractive.

C. Bi-LSTM

RNNs treat inputs dependently. They perform the current task, such as predicting a word by examining dependencies across recent information. However, they only can capture the dependencies of the *i*th word x_i given the previous words, such as $x_{i-3}:x_{i-1}$. For many NLP tasks, checking the previous information is generally insufficient to perform an accurate prediction, the subsequent words, such as $x_{i+1}:x_{i+3}$ of word x_i, can also be useful. To obtain the dependencies of the surrounding words of word x_i, the bidirectional RNN (Bi-RNN) [21] is designed to serve this purpose. The Bi-LSTM, which is a variation of Bi-RNN, is similar in essence but captures longer dependencies.

Contextual information is crucial for vulnerability detection. Because the source code is a logical and semantic structure, it is closely connected and tightly coupled. Hence, the occurrence of a vulnerable code fragment is usually related to either previous or subsequent code, or even to both. A vulnerable code fragment usually contains multiple lines of code, which can be distributed across a function block. In many cases, it is difficult to exactly pinpoint which line of code actually causes the vulnerability. Hence, the bidirectional implementation of LSTM can help to detect a long-term dependency of both forward and backward, which can effectively capture the vulnerable programming patterns.

D. Pretraining for Obtaining Representations

The pretraining phase trains a basic network using the historical vulnerability data of several different software projects. These projects (a.k.a source projects) initialize the network parameters for learning low-level features of vulnerabilities. As depicted in Fig. 4, two dense layers are added after the global max pooling layer to form a complete network. The training inputs are AST-based features from both vulnerable and nonvulnerable functions. This makes sure that the hidden nodes capture the sequential interactions that are discriminative of vulnerable programs. The input data are divided into training and validation sets to build and evaluate the model and guide the model tuning processes to maximize the performance. Once the model is trained and the performance is satisfactory, we feed the trained networks with the processed AST-based features of a target project with limited labels and obtain the learned representations from the third layer of the networks. Given a sequence of an arbitrary length as an input fed to the networks, the learned representation is a 128-dimensional vector.

IV. Performance Evaluation

A. Experiment Datasets

To the best of our knowledge, there is no publicly available, function-level vulnerability ground truth dataset. Although previous studies [25], [24], and [9] have focused on vulnerability detection at the function-level granularity, their data are not publicly accessible. Therefore, we constructed a function-level vulnerability dataset from scratch and have made our data and code publicly available on GitHub.[1] We manually labeled 457 vulnerable functions and collected 32 531 nonvulnerable functions from six open-source projects across 1000+ popular releases. We obtained the vulnerability data from the National Vulnerability Database (NVD) and the common vulnerability and exposures (CVE). Both NVD and CVE repositories use CVE ID as unique identifiers for vulnerabilities. CVE IDs are assigned to vulnerabilities, allowing a security professional to

[1]https://github.com/DanielLin1986/TransferRepresentationLearning

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SOURCE CODE PROJECTS INVOLVED IN EXPERIMENTS

Project	# Vulnerable Functions Labeled	# Non-vulnerable Functions Used
LibTIFF	96	777
LibPNG	43	499
FFmpeg	191	4921
Pidgin	29	8050
VLC Media Player	42	3636
Asterisk	56	14648

quickly access the technical information of known vulnerabilities across multiple CVE-compatible sources. NVD offers a convenient option for searching the known vulnerabilities of a software project. Using the NVD description, we can download the corresponding version of a project and locate each vulnerable function in the software project's source code and label it accordingly.

Our experiments included six open-source projects:

1) LibTIFF;
2) LibPNG;
3) FFmpeg;
4) Pidgin;
5) VLC Media Player;
6) Asterisk.

Their source code can be obtained from GitHub or their public code base. For each of these projects, we manually labeled the vulnerable functions recorded on CVE and NVD until October 1, 2017. Then, excluding the identified vulnerable functions, we selected the remaining functions as nonvulnerable ones. Table I provides a summary of the related projects and the number of functions used in our experiments.

B. Environment and Parameters

The implementation of the Bi-LSTM network used Keras (version 2.0.8) [3] with TensorFlow (version 1.3.0) [1] backend. The random forest algorithm was provided by the scikit-learn package (version 0.19.0) [19]. The Word2vec embedding was provided by the gensim package (version 3.0.1) with all default settings. The computational system is a server running CentOS Linux 7 with two Physical Intel(R) Xeon(R) E5-2690 v3 2.60-GHz CPUs and 96 GB of RAM.

When mapping ASTs to vectors, we made a tradeoff between the tree complexity and a shallow representation. Since a complex AST results in a long vector containing thousands of elements, we need to truncate overcomplicated ASTs when converting them to vectors of the same length for balancing between information loss and excessively long vectors. We observed that approximately 93% of AST samples are within 1000 elements in length, so, we truncated the vectors which have more than 1000 elements, and for the ones having fewer elements, we padded them with 0 s.

Table II shows the parameters we tuned for pretraining the Bi-LSTM network with source projects. Then, the trained network was fed with the target project for generating representations that were used as features for training a random forest classifier

TABLE II
TUNED PARAMETERS FOR TRAINING BI-LSTM NETWORKS

Parameter	Description (value[1])
Embedding_dim	The dimensionality of embedding vectors that the elements of AST vectors will be converted to (100).
Data_dim	The dimensionality of the input vector (1000).
Bi-LSTM Units	The number of the Bi-LSTM units per layer (64).
Batch_Size	The number of training samples that are propagated through the network at a time (32).
Epoch	One forward/backward pass of all the training samples (150).
Monitor	The training will stop if the validation accuracy ceases to increase (val_acc).
Loss function	The choice of loss function to minimize (binary_crossentropy).
Optimizer	The RMSprop with default parameters was used.
Activation	The tanh function is used for hidden layers and the sigmoid function is used for the last layer (tanh, sigmoid).

[1] These values were used in Keras for implementing the Bi-LSTM networks.

to acquire a list of functions ranked based on their probabilities of being vulnerable.

C. Evaluation Metrics and Baseline

The performance of our method is measured by the proportion of vulnerable functions returned in a function list. Hence, the metric that we apply for performance evaluation is the top-k precision (denoted as $P@K$). The metric is usually used in the context of information retrieval systems, such as search engines, for measuring how many relevant documents are acquired in all the top-k retrieved documents [5]. In our context, the P@K refers to the proportion of vulnerable functions in the top-k retrieved functions. In our experiments, k ranged from 10 to 200 to simulate a practical case where not all code is audited due to time and resources limitations.

We compared our method with the approach of applying the traditional CMs, since CMs are quality measures for quantifying programs' complexity in software tests. Therefore, they are important indicators of software faults and vulnerabilities, as complex code is difficult to comprehend and, therefore, hard to maintain and test, which might introduce faults and vulnerabilities into software systems [4] and [22]. Using Understand, a commercial code enhancement tool, we were able to collect function-level CMs from source code. We randomly selected 23 CMs (i.e., lines of code, cyclomatic complexity, essentials, etc.) as features for vulnerability detection to compare with our method using the AST representations as features. Both feature sets were trained separately with a random forest classifier for comparison.

Other metrics adopted for measuring the effectiveness of a prediction model in code auditing include the reduced amount of code or files to inspect, compared to not using the model. Such metrics were used in [22] as estimators of cost reduction. In our scenario, to quantify the inspection reduction in terms of functions, we define the function inspection reduction rate (FIRR) for measuring how much effort can be saved with our method compared with the method using CMs as features. The FIRR is the ratio of the reduced number of functions needed for inspection to the number of functions that are randomly

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.

selected. Given a recall value achieved by our method, e.g., 60%, one needs to randomly select 60% of the total number of functions in order to achieve the same recall. Therefore, with a recall value achieved by a given method, the amount of functions that a random selection needs to select (denoted as N_{random}) can be calculated as

$$N_{\text{random}} = \text{recall} \times N_{\text{total}} \tag{1}$$

where the N_{total} is the total number of functions used for testing.

Given a recall achieved by using CMs, to acquire the same amount of vulnerable function, the number of functions (denoted by N_{CM}) that random selection needs to select can be calculated as

$$N_{\text{CM}} = \text{recall}_{\text{CM}} \times N_{\text{total}}. \tag{2}$$

Given a recall achieved by applying deep AST representations, to acquire the same amount of vulnerable function, the number of functions (denoted by N_{AST}) that random selection needs to select can be calculated as follows:

$$N_{\text{AST}} = \text{recall}_{\text{AST}} \times N_{\text{total}}. \tag{3}$$

Therefore, the FIRR of using AST-based representation to CM-based features can be defined as

$$\text{FIRR}_{\text{AST-to-CM}} = \frac{N_{\text{AST}} - N_{\text{CM}}}{N_{\text{AST}}} = 1 - \frac{N_{\text{CM}}}{N_{\text{AST}}}. \tag{4}$$

D. Experiment Settings and Outcomes

1) Deep AST Representations Versus CMs, and Random Selection: We first used the FFmpeg project to evaluate the effectiveness of our AST-based features generated using the proposed approach. We collected 4921 functions across multiple releases of the FFmpeg project, in which there are 191 vulnerable functions. We used our proposed feature processing technique to process the collected samples and divide them into three datasets: 1) training; 2) validation; and 3) testing, with a ratio of 13:4:3. The training and validation sets are used to train the proposed Bi-LSTM network. Then, the performance is evaluated on the testing set. For comparison, we combined the previous divided training and validation sets as the new training set and use the 23 CMs as features for training a random forest classifier. Subsequently, we validated the trained classifier on the same testing set.

The testing set contains 741 functions, among which there are 25 vulnerable samples. Fig. 5 shows that the green line, which is the precision achieved by using deep AST representations lies above the blue line being the precision obtained using CMs as features. When retrieving the 10 most probable vulnerable functions, eight vulnerable functions were identified using our method. With CMs, only five were found. Although when searching for the 20 most probable vulnerable functions, both methods identified nine functions, when retrieving more functions, our method achieved higher precision. The reason was that using CMs as features helped to pick the vulnerable functions which were long and complex. Within the top 20 functions, there are some long vulnerable functions which were found by both approaches. As more functions were examined, the CM-based

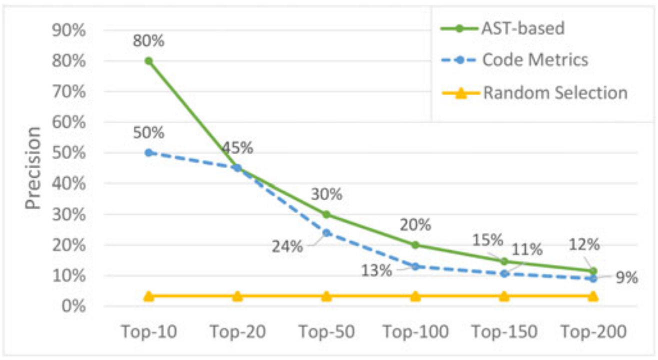


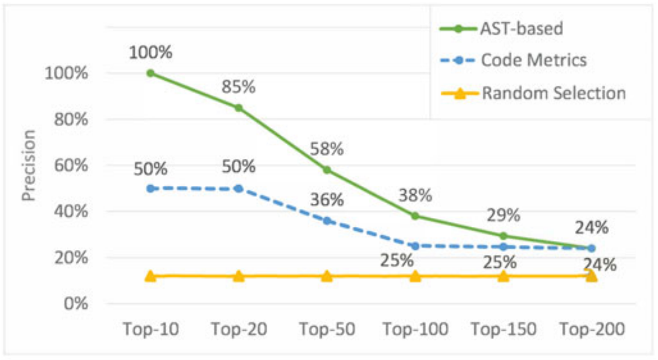
Fig. 5. Precision comparison between deep AST representations (AST-based), CMs, and random selection on FFmpeg.

features would generate more false positives, as almost all of the long functions were recognized as vulnerable by CMs. When examining the top 200 most probable vulnerable functions, our method found 24 out of 25 actual vulnerable functions, while with CMs, only 18 were identified and for random selection, one could only find six vulnerable functions by chance. When the top 500 most probable vulnerable functions were returned, both AST-based and CMs methods achieved 100% recall.

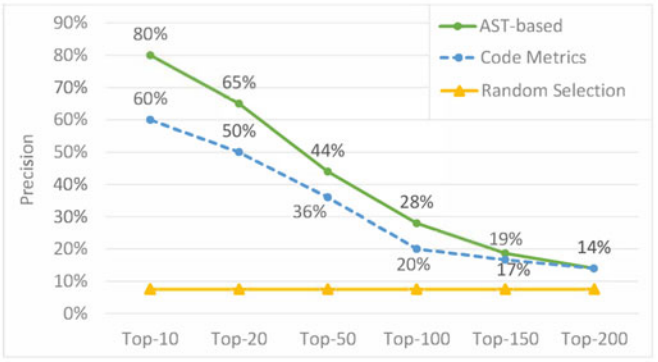
2) Transfer-Learned Representations Versus CMs, and Random Selection: We conducted three experiments to demonstrate the effectiveness of the proposed approach, which helps to apply learned knowledge from the source projects to the target project for vulnerability detection. Among the six projects listed in Table I, we chose one project as the target project, and the other five as the source projects. We used the functions from source projects to train a network, and with the trained network, we fed it with the target project code to obtain the deep AST representations. Then, we further divided the target project into training and testing parts with a 1:3 ratio to simulate the scenario that there usually is insufficient labeled data for vulnerability discovery. Last, for the training and testing parts, we used the obtained representations and CMs to train random forest classifiers, respectively.

Fig. 6(a) shows the comparison results of our method, CM-based features, and random selection on project LibTIFF. The testing set contains 583 functions among which there are 71 vulnerable ones. When examining the top 10 functions, our approach achieved 100% precision. With 20 most probable vulnerable function retrieved, 17 vulnerable functions were found. Fig. 6(b) illustrates the comparison results on LibPNG with 347 nonvulnerable functions and 28 vulnerable functions. With our approach, achieving 28% precision means that checking the top 100 most probable vulnerable functions was able to identify all of the vulnerable functions. However, with CM-based features, one needs to examine the top 200 function list to achieve the same goal. Similar to the results on LibTIFF, the test on project FFmpeg shows that 100% precision was achieved on the top 10 function list, as shown in Fig. 6(c). On the testing dataset for FFmpeg, there are 143 vulnerable functions out of 3691 total number of functions. Despite the imbalance between vulnerable and nonvulnerable samples, checking around 5% of the total

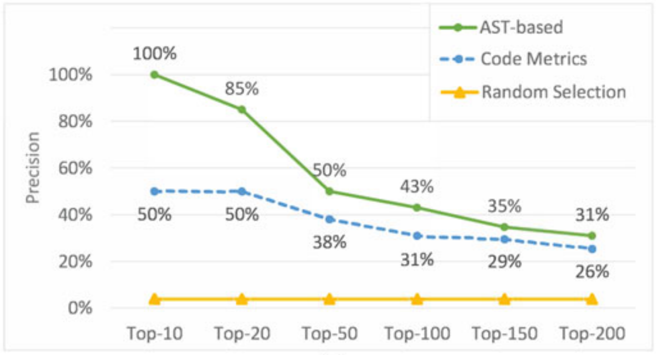
Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.



(a)



(b)



(c)

Fig. 6. Precision comparison among the transfer-learned representations (AST-based), CMs, and random selection on different projects. The general trend in the three figures shows that the proposed method using transfer-learned representations as features gained the best precision. (a) Testing on LibTIFF. (b) Testing on LibPNG. (c) Testing on FFmpeg.

number of functions can discover 43% of vulnerable samples with our approach, while using CM-based features, one can only discover 36% of vulnerable functions. Hence, the results showed that the transfer-learned deep AST representations were more effective than the human-defined CMs on our datasets. However, we also observed that when retrieving 500 functions both AST-based and CMs methods can identify all the vulnerable functions.

3) Measuring Function Inspection Reduction: The FIRR provides an estimator for measuring the cost and effort from a perspective of the reduction of the overall functions needed for code inspection since the purpose of adopting the ML technique is to minimize cost and human effort. Fig. 7 shows the overall trend of FIRR of using deep AST representations to applying CM as features on three projects. Generally, compared with the method using CM as features, our approach can significantly reduce the number of functions for inspection when retrieving fewer than 100 functions. Among three projects, our approach performed well on FFmpeg, as a maximal reduction of 57% of functions for inspection was observed. But, on LibPNG, the number of reduced functions was not as large as that on the other projects. Our future work will further investigate how project differences affect the performance of our approach.

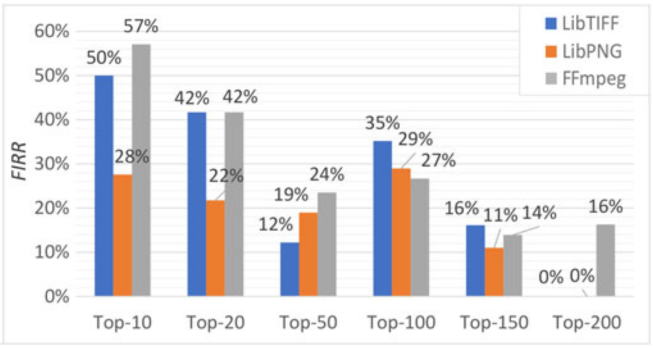


Fig. 7. FIRR of transfer-learned AST representations to CMs as features.

E. Discussion

Our proposed approach takes source code functions as inputs, which facilitates the filtering of potentially vulnerable functions during a development process. However, for vulnerabilities that involve multiple functions or multiple files, our method is not directly applicable. This will be addressed in our future research.

When applying the LSTM network for representation learning, the training process on 23 000+ samples took up to 6 h on our server. In practice, this problem can be mitigated by training the network offline so that the trained network is ready for extracting the learned representations.

Another challenge arises from the severe imbalance between any two classes which affects the detection performance. To overcome this, we apply a random forest classifier which is an ensemble classifier for training on the learned representations. We believe that the detection performance can be further improved by addressing the imbalance issues with techniques, such as oversampling or undersampling. Additionally, the empirical results showed that some vulnerable functions are in the top 50 retrieved list detected by using features in CM but not in the top 50 list that using AST representations. It will be an interesting research direction to combine the two methods for better identification of vulnerable functions.

V. Conclusion

Our framework leverages ASTs and their deep representations to convert project-specific source code to project-agnostic features capturing deeper structures and semantics of functions. This allows vulnerable programming patterns learned from soft-

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.

ware source projects to facilitate the representation generation on a target project for better vulnerability prediction. We first learn the deep AST representations, then we process the ASTs by converting them to sequences of elements. Following this, the ASTs are tokenized and mapped to vectors that preserve code semantics. To concisely capture the rather complex sequential interactions among different program segments, we propose a deep-learning network architecture to learn high-level abstractions, which we refer to as deep AST representations. Trained with historical software vulnerability data, these representations become refined features reflecting the intrinsic patterns indicative of a software vulnerability. Finally, these learned features are fed to train a random forest algorithm and to obtain a ranking list showing the most probable functions that are vulnerable.

References

[1] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," *Operating Syst. Des. Implementation*, vol. 16, pp. 265–283, 2016.

[2] C. Cadar *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *Operating Syst. Des. Implementation*, vol. 8, pp. 209–224, 2008.

[3] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://github.com/fchollet/keras

[4] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, 2011.

[5] P. R. Christopher, D. Manning, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[6] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Syst. Rev.*, vol. 35, no. 5, pp. 57–72, 2001.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[8] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire OS distributions," in *Proc. 2012 IEEE Symp. Security Privacy*, 2012, pp. 48–62.

[9] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. 2017 IEEE Symp. Security Privacy*, 2017, pp. 595–614.

[10] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: An automated vulnerability detection system based on code similarity analysis," in *Proc. 32nd Annu. Conf. Comp. Security Appl.*, 2016, pp. 201–213.

[11] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. 2017 ACM SIGSAC Conf. Comput. Security*, 2017, pp. 2539–2541.

[12] A. Meneely and L. Williams, "Secure open source collaboration: An empirical study of Linus' law," in *Proc. 16th ACM Conf. Comput. Commun. Security*, 2009, pp. 453–462.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. ICLR workshop*, 2013.

[14] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010, pp. 1045–1048.

[15] L. Moonen, "Generating robust parsers using island grammars," in *Proc. IEEE 8th Working Conf. Reverse Eng.*, 2001, pp. 13–22.

[16] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th Conf. Comput. Commun. Security*, 2007, pp. 529–540.

[17] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of NDSS*, 2005.

[18] C. Olah, "Understanding LSTM networks," *GITHUB Blo*, vol. 27, Aug. 2015. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[19] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[20] H. Perl *et al.*, "VCCfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 426–437.

[21] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[22] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov.–Dec. 2011.

[23] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson Education, 2007.

[24] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX conference on Offensive technologies*, USENIX Association, 2011, p. 13.

[25] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comp. Security Appl. Conf.*, 2012, pp. 359–368.

[26] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. 2013 ACM SIGSAC Comp. Commun. Security*, 2013, pp. 499–510.

Authors' photograph and biography not available at the time of publication.

Authorized licensed use limited to: ON Semiconductor Inc. Downloaded on November 24,2022 at 12:33:29 UTC from IEEE Xplore. Restrictions apply.