Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding

Xiao Cheng[1], Haoyu Wang[1], Jiayi Hua[1], Miao Zhang[1], Guoai Xu[1], Li Yi[2] and Yulei Sui[3]
[1] Beijing University of Posts and Telecommunications, Beijing, China
[2] National Computer Network Emergency Response Technical Team Coordination Center, Beijing, China
[3] University of Technology Sydney, Australia

***Abstract*—Static vulnerability detection has shown its effectiveness in detecting well-defined low-level memory errors. However, high-level control-flow related (CFR) vulnerabilities, such as insufficient control flow management (CWE-691), business logic errors (CWE-840), and program behavioral problems (CWE-438), which are often caused by a wide variety of bad programming practices, posing a great challenge for existing general static analysis solutions. This paper presents a new deep-learning-based graph embedding approach to accurate detection of CFR vulnerabilities. Our approach makes a new attempt by applying a recent graph convolutional network to embed code fragments in a compact and low-dimensional representation that preserves high-level control-flow information of a vulnerable program. We have conducted our experiments using 8,368 real-world vulnerable programs by comparing our approach with several traditional static vulnerability detectors and state-of-the-art machine-learning-based approaches. The experimental results show the effectiveness of our approach in terms of both accuracy and recall. Our research has shed light on the promising direction of combining program analysis with deep learning techniques to address the general static analysis challenges.**

***Index Terms*—Static analysis, graph embedding, vulnerabilities, control-flow**

I. Introduction

Modern software is large and complex widely used in all aspects of our daily life, from desktop applications to mobile apps, from embedded systems to data centres. Unfortunately, complex system software is often plagued with vulnerabilities, resulting in serious reliability and security concerns. Automatic detection of software vulnerabilities through software analysis and testing has become a fundamental approach to taming these reliability and security issues.

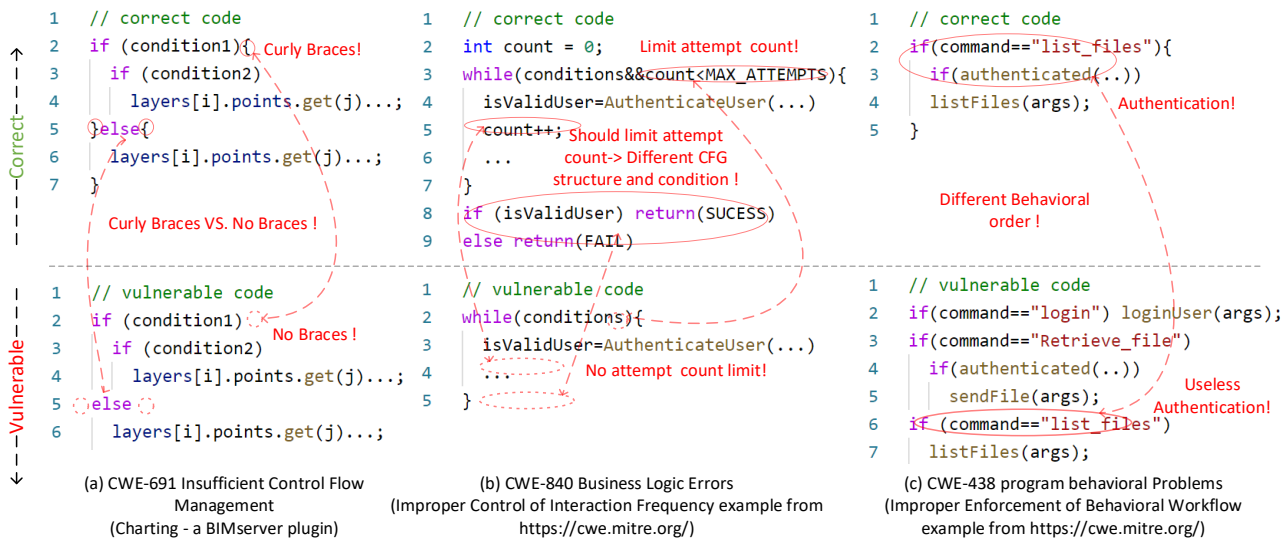
Static bug detection, which approximates the runtime behaviour of a program without running it, is the major way to pinpoint bugs at the early stage of software development cycle, thus reducing software maintenance cost. The existing static analysis techniques (e.g., Coverity [3], Fortify [5], Flawfinder [4], ITS4 [30], RATS [6], Checkmarx [2] and SVF [29]) have shown their successes in detecting traditional vulnerabilities (e.g., buffer overflows, memory leaks and use-after-frees). However, these approaches that rely on conventional static analysis theories (e.g., data-flow and abstract interpretation) are still ineffective in detecting non-traditional control-flow-related (CFR) bugs, such as insufficient control flow management (CWE-691), business logic errors (CWE-840), and program behavioral problems (CWE-438), which are often caused by bad programming practices.

Figure 1 shows three real-world examples of the aforementioned three CFR vulnerabilities. The code fragment in Figure 1(a) shows a CWE-691 vulnerability found from `Charting`, a BIMserver plugin that can create all sorts of charts. The keyword 'else' should match the first 'if', but not the second one. However, the developer forgot the curly braces, resulting in a wrong execution scope on the control-flow. Figure 1(b) shows a code fragment from CWE™ [9]. The vulnerable code does not place any restriction on the number of authentication attempts made, which leaves attack surfaces that can be leveraged by attackers to launch brute force attempts. This shows an anomaly code pattern that causes a logic error due to the missing of conditional checks. The code fragment in Figure 1(c) is from a FTP server. The correct code requires that users must successfully login before performing any other action such as retrieving or listing files. However, the vulnerable code which is different from other correct authentication code snippets in the same project (in terms of control-flow orders), will incorrectly list the files without confirming users' identities.

Unlike low-level memory errors which have clear and well-defined bug specifications for static analysis (e.g., a use-after-free happens when referencing an object that has been freed), high-level CFR bugs are often triggered due to bad yet complicated programming practices, posing a great challenge for existing rigours and traditional static analyzers in the presence of a wide variety of anomaly code patterns.

It is non-trivial for human experts to define customized rules for detecting CFR vulnerabilities. This is not only because vulnerability detection itself is difficult in a complex software system, but also because different patterns may require different detecting rules. The quality of each rule also varies with individuals. Thus, the results are often limited by their existing experience in summarizing vulnerability patterns. Simply designing an unsound analysis using user-defined pattern matching may result in a large number of false positives and/or false negatives.

Recently, machine learning has shown its effectiveness in boosting the performance of static analysis for detecting memory errors such as buffer overflows [13], [21] and use-after-frees [35]. However, many state-of-the-art static approaches require compilable and complete source code, which makes hard for analyzing incomplete or incompilable code fragments



(a) CWE-691 Insufficient Control Flow Management (Charting - a BIMserver plugin)

(b) CWE-840 Business Logic Errors (Improper Control of Interaction Frequency example from https://cwe.mitre.org/)

(c) CWE-438 program behavioral Problems (Improper Enforcement of Behavioral Workflow example from https://cwe.mitre.org/)

Fig. 1. Real-world control-flow-related (CFR) vulnerabilities that are hard to be automatically identified by traditional static vulnerability detection approaches. For each example, we list the correct code fragment and the corresponding identified vulnerable code.

under real-world settings, particularly for detecting high-level CFR vulnerabilities. It remains open as to whether we can leverage machine learning or its promising branch, deep learning, to produce an accurate code representation to support detecting complicated high-level CFR vulnerabilities.

In this paper, we propose VGDETECTOR, a new deep-learning-based code embedding approach to detecting control-flow-related (CFR) vulnerabilities. Our approach makes a new attempt by applying a recent graph convolutional network to embed code fragments in a compact and low-dimensional code representation that preserves high-level control-flow information of a vulnerable program, without the needs of manually defined rules. We have conducted our experimental evaluation based on a large-scale benchmark harvested from the Software Assurance Reference Dataset (SARD) [8], which contains a set of known real-world security flaws. To validate the effectiveness of our approach, we have compiled a list of 8,368 real-world programs (roughly 60K methods) which are only close related to CFR bugs (i.e., CWE-691, CWE-840 and CWE-438). To demonstrate the effectiveness of our approach, we have conducted extensive experiments by comparing our approach with both well-known conventional static detectors (including Flowfinder [4] and RATs [6]) and state-of-the-art machine-learning-based approaches (including Token-based embedding [33]) and Vuldeepecker [22])).

The key contributions of this paper are as follows:

- We propose VGDETECTOR, a new code representation approach using graph convolutional network to embed control-flow information of vulnerable code fragments that do not have to be compilable.
- We present new insights and an effective deep-learning-based approach to detect high-level CFR vulnerabilities, i.e., insufficient control flow management (CWE-691), business logic errors (CWE-840), and program behavioral problems (CWE-438). To the best of our knowledge, no previous studies have investigated such kinds of vulnerabilities and provided effective solutions.
- Since there are no readily available reference datasets for complicated high-level CFR vulnerabilities, we also contribute a large-scale reference dataset to our community from real-world programs to boost future research studies.
- We have conducted our experiments by comparing VGDETECTOR with the traditional static vulnerability detectors and two recent machine-learning-based approaches in characterizing CFR vulnerabilities. The results show the effectiveness of our deep-learning-based approach in terms of both accuracy and recall.

II. DESIGN OF VGDETECTOR

The goal of VGDETECTOR is to automatically and statically detect CFR vulnerabilities without the knowledge of well-defined patterns of such vulnerabilities, thus saving the efforts of human inspection to identify high-level yet complicated vulnerabilities. To this end, we propose a deep-learning-based approach, and the key idea is to first perform program analysis on the source code to extract control-flow related semantic features, and then combine them with graph convolutional network to embed code fragments in latent feature space.

This section first introduces the design overview of VGDETECTOR. We then detail the major components of VGDETECTOR including *code pre-processing*, *control flow graph (CFG) construction*, *CFG embedding* and *neural network training*.

A. Overview of VGDETECTOR

As shown in Fig. 2, the overall inspection process is divided into two phases, the training and detecting phases. The inputs of both phases are the source code of programs. A neural network

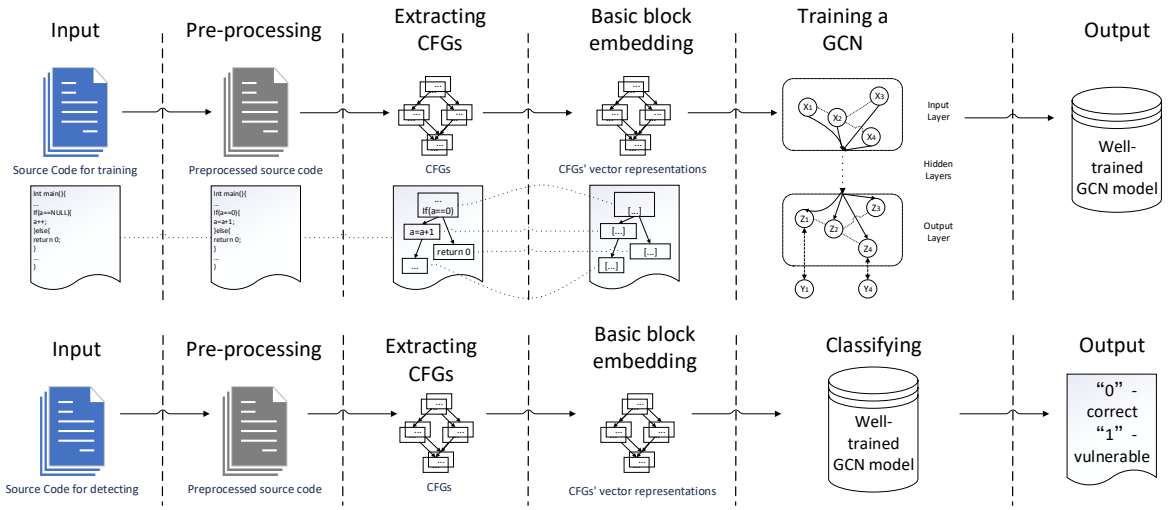


Fig. 2. Overview of VGDETECTOR. The training phase (on the top side) aims to train a model encoded with vulnerability patterns and the detecting phase (on the bottom side) is to flag whether the target code is vulnerable or not.

model is established in the training phase, embedded with the control-flow related information of vulnerable code fragments, reflecting the high-level patterns of CFR vulnerabilities for later accurate vulnerability detection.

1) Training phase: Our training phase consists of four main steps, including (1) pre-processing, (2) control-flow graph construction and labeling, (3) embedding for each basic block on the constructed CFG using `Doc2Vec` [18], and (4) the overall CFG embedding using graph convolutional network [17] on top of the `Doc2Vec` embedding of the nodes on the CFG. VGDETECTOR works directly on the source code of a program instead of a compiled intermediate representation, thus our approach is able to support embedding of incomplete or uncompile code fragments. Embedding a basic block via `Doc2Vec` treats its statements as a sequence of tokens, which needs to be pre-processed to produce a more canonical form of the source code for an accurate `Doc2Vec` embedding, while keeping changing the underlying semantics of the code.

Step 1: Source code pre-processing. For the source code that preserves the same semantics, there may be multiple ways for implementation, depending on the coding style of developers, e.g., naming conventions, statement expressions, etc. Moreover, developers may define macros in the source code. Thus, we seek to unify the code that preserves the identical semantics by summarizing a list of transformation rules (cf. Table I) and expanding the defined macros if available. The goal of this step is to eliminate the inconsistencies in the source code representation, thus to achieve accurate word embedding.

Step 2: Constructing the control flow graph. Since we aim to detect CFR vulnerabilities, the control-flow-related semantic information (e.g., CFG together with its branch conditions) is the major choice for the representation of the source code. Thus, we first generate the CFG for each function based on its AST generated by the tool ANTLR4 [1] (more details are discussed in Section II-C). Each constructed CFG is further flagged as whether it contains CFR vulnerabilities or not.

Step 3: Basic block embedding via `Doc2Vec`. A neural network takes a vector as an input, thus we first transform each basic block in the constructed CFG, including its statements and branch conditions, into vector representation. Here, we take advantage of `Doc2Vec` [18] for embedding code (in the form of tokens) into fixed length low-dimensional vectors.

Step 4: Graph embedding via GCN. To accurately embed graph data structure such as CFG, it is hard to apply neural models like CNNs and RNNs, which perform well on sequential tokens but not on graphs. We use graph convolutional neural networks [17], which represent a recent advance in precise graph embedding to perform classic ML tasks, such as classification and clustering. We use the vector representation of each basic block of a CFG as the feature of each node. Given the edges of the CFG in the form of adjacency matrix and the feature of each node in the CFG, we can train our model by using graph convolutional neural networks [17] to conduct standard classification of nodes on the graph.

2) Detecting phase: The detecting phase performs CFR vulnerability prediction for a target program using the trained model by feeding the extracted CFG related features. Similar as the training phase, the CFGs of the target program are first constructed and embedded into a latent space after pre-processing. We then perform classification for detecting anomaly code fragments that may contain CFR vulnerabilities.

B. Pre-processing source code

Our approach analyzes a program at the source code level to handle code fragments that may be incomplete or uncompilable. Therefore, we have adopted a pre-processing step first to produce a more canonical form of the original source code. As aforementioned, the different representations of the same semantics (e.g., expression "a + = 1" is equivalent to "a = a + 1")

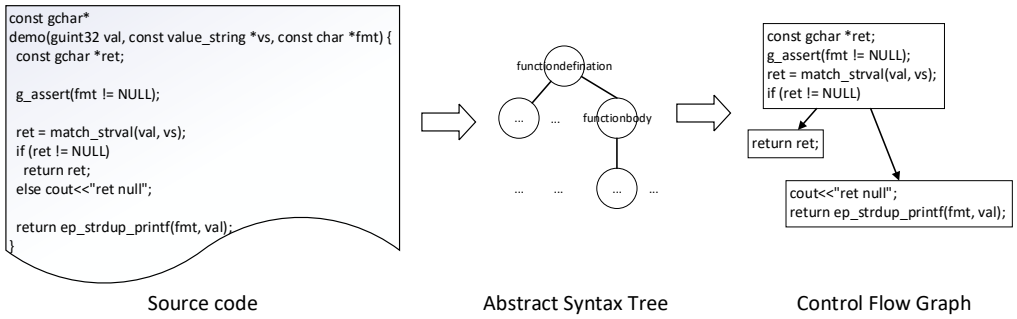


Fig. 3. An Example of Constructing the Control Flow Graph.

lead to different orders of code tokens, making `Doc2Vec` [18] embedding of basic block imprecise.

Inspired by the text pre-processing step in the NLP-related studies, we transform the code text before the `Doc2Vec` [18] embedding step. We first convert all the letters in the source code to the lower case and remove punctuations, accent marks, and other diacritics. This eliminates the inconsistency of code recognition caused by different coding styles. We try to unify the code while preserving the identical semantics by summarizing a list of rules belonging to four main categories (Table I), including "removing operations", "simplifying operations", "replacing operations", and "other operations". Furthermore, we have expanded the defined macros in the source code if available. Here, we take advantage of the `Tscancode` [7] to perform the aforementioned transformations.

C. Constructing the CFG

We first generate the abstract syntax trees (ASTs) of a program by taking advantage of ANTLR4 [1] (Another Tool for Language Recognition), which is a robust parser generator for reading, processing, executing, or translating structured text or binary files. Then we traverse the ASTs to identify the contained functions. Next, we construct the CFG for each function, on which each node represents a basic block (i.e., a unit of straight-line statements with no branches), and each edge signifies the control-flow between two basic blocks, representing the possible program execution order. This representation together with the branch conditions are very useful for pinpointing CFR vulnerabilities. Fig 3 shows an example of the constructed CFG for the given code fragment. Note that we perform vulnerability prediction at the *granularity of method (CFG)*, i.e., each method is flagged as vulnerable or not by referring to the ground truths.

D. Basic Block Embedding

For each constructed CFG, we first transform it into the symbolic representation. As shown in Figure 4, we have replaced all the user-defined variables and functions with their symbolic names (e.g., variables are with names "VAR1" and "VAR2", functions are with names "FUN1" and "FUN2") in a one-to-one mapping manner. It is worth mentioning that multiple variables/functions may be mapped to the same symbolic name when they appear in different CFGs.

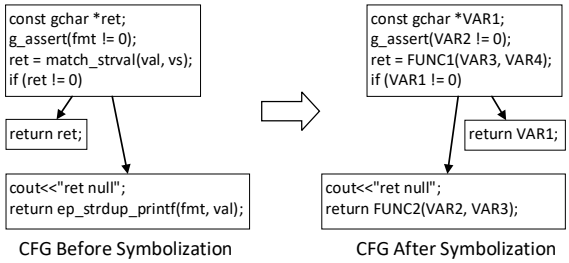


Fig. 4. Transforming a CFG to its Symbolic Representation.

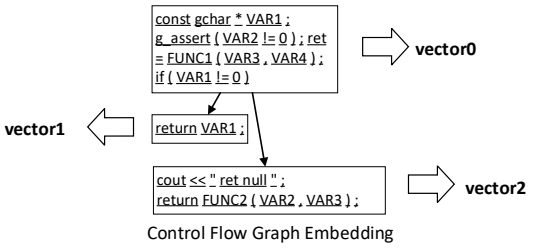


Fig. 5. Embedding CFGs' Symbolic Representations Into Vectors.

Then, we will encode each basic block on the CFG into vectors using `Doc2vec` [18], a widely used tool to represent documents as fixed length low-dimensional vectors (i.e. document embeddings). Fig 5 shows an example of the basic block embedding process. Each basic block is considered as a document, consisting of several lines of statements. We first tokenize the document and encode it into a vector using document embedding method named Distributed Memory version of Paragraph Vector (PVDM). It acts as a memory that remembers what is missing from the current context — or as the topic of the paragraph. The document vector intends to represent the concept of a document (i.e. basic block).

E. Graph Embedding

Previous work [17] has suggested that the graph convolutional neural network performs well on traditional machine learning tasks with graph data structures. Here, we choose GCNConv [17] as the convolutional layer of our model, which is scalable for semi-supervised learning on graphs. This model

TABLE I
FOUR TYPES OF RULES FOR PRE-PROCESSING THE SOURCE CODE.

Operation Catagory	Detailed Description
Removing operations	MACRO in variable declaration like "MACRO int x" redundant parentheses keywords('deprecated','volatile','inline','register','restrict') expr.("extern 'C"' and "extern 'C"' {}) calling conventions("__cdecl"," __stdcall"..) expr.("__attribute__((?))" and "__declspec()")
Simplifying operations	C alternative tokens ('and','or',etc.) simple calculations "0[foo]" => "*(foo)" case ranges (gcc extension) labels and "caseldefault"-like syntaxes "[;{}]({code;});"=>"[;{}] code;"
Replacing operations	inline SQL => "asm()" "a ## b" => "ab" 'NULL' and similar '0'-defined macros => '0' platform dependent types => standard types (32 bits: size_t -> unsigned long ;64 bits: size_t -> unsigned long long) "unsigned long long int" => "long" (with "_isUnsigned=true","_isLong=true") 'sin(0)' => '0' and other similar math expressions "x = ({ 123; });" => "{ x = 123; }" operator name tokens => single token("operator =" => "operator=") "a+=b;" => ";a=a+b;" "f(x=g())" => "x=g(); f(x)" e.g."atol("0")" => '0'
Other operations	expand typedef and user-defined macros Combine strings and "- %num%" Split up variable declarations Handle templates Put "^ {}" statements in "asm()" Order keywords 'static' and 'const'

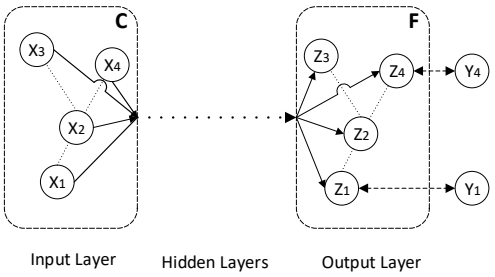


Fig. 6. The structure of the graph convolutional neural network.

is based on an efficient variant of the convolutional neural network. Fig. 6 shows an example of the structure of the GCN for semi-supervised learning with C input channels and F feature maps in the output layer.

Fig 7 shows the structure of our neural network. We choose the node feature matrix and the adjacency matrix of the control flow graph as the input of our neural network. Then the two matrices were feed to the convolutional part. The convolutional part consists of a convolutional layer and max-pooling layer. The activation function we chose in our system is the Rectified Linear activation function (ReLU). In the convolutional layer, features were extracted from the node feature matrix and the adjacency matrix of the input graphs. After each convolutional layer, we use max-pooling to reduce the feature dimensions. In our detection system, there can be multiple layers of the convolutional part. We use a hidden fully connected layer after the last convolutional layer, which allows high-order relationships between the features to be detected. Finally, a softmax layer is used to output the probabilities of class labels. It is worth noting that most of the parameters in the neural network are updated automatically by back propagation during training. Next, we will introduce the propagation rule of convolution layer in detail as it is the most important part.

Propagation rule of the convolutional layer. The purpose of the convolution is to extract different features of the inputs. More convolutional layers of the network can iterate extracting more complex features from low-level features. In our proposed architecture, there can be multiple convolutional layers being used. These convolutional layers are numbered from 1 to $\mathbf{L}$. The first convolutional layer's input is the node feature matrix $\mathbf{X}$, with shape $[num\_nodes, num\_node\_features]$, where num_nodes is the total number of the basic blocks and $num\_node\_features$ is the dimension of the vector representation of each basic block. We consider a multi-layer Graph Convolutional Network (GCN) with the following layer-wise propagation rule:

$$\mathbf{H}^{(l+1)} = \sigma\left(\widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{1}$$

$\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ represents the adjacency matrix of the graph with added self-connections. Among them $\mathbf{I}_N$ is the identity matrix. $\widetilde{\mathbf{D}}_{ii} = \sum_j \widetilde{\mathbf{A}}_{ij}$ and $\mathbf{W}^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes an activation function. We choose Rectified Linear activation function(ReLU) as our activation function:

$$ReLU(\cdot) = max(0, \cdot) \tag{2}$$

$\mathbf{H}(l)\, \epsilon \mathbf{R}^{N \times D}$ means the matrix of activations in the l^{th} layer of the neural network. For the first layer, $\mathbf{H}^{(0)} = \mathbf{X}$. As can be seen from the propagation rule, the adjacency matrix of the graph and the feature of each node are encoded into the convolutional layer. It means that our model scales linearly in the number of control flow graph edges. The hidden layer representations is learned by encoding both local control flow graph structure and the features of each basic block.

III. EXPERIMENTAL SETUP

To the best of our knowledge, no available benchmarks regarding CFR vulnerabilities could be applied to our evaluation directly. Thus, we first propose to harvest a reliable dataset of

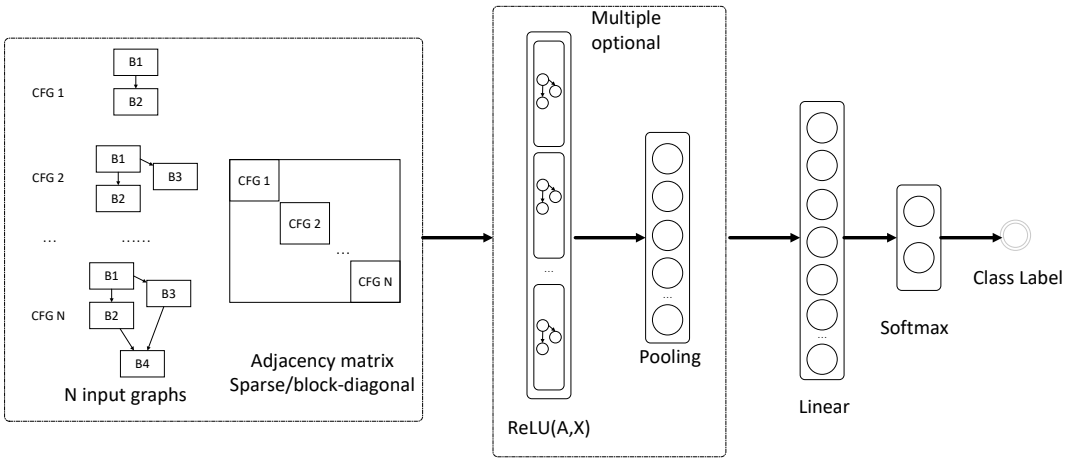


Fig. 7. The structure of our neural network.

CFR vulnerabilities from real-world programs. Then, we detail the training process of the proposed GCN network.

A. Creating the benchmark

We have harvested a comprehensive CFR vulnerability benchmark dataset from Software Assurance Reference Dataset (SARD) [8], which hosts a large number of known real-world security flaws. It is widely used to evaluate the performance of vulnerability detection approaches in our community (e.g. [11] [22] [10].) In the SARD dataset, each program (i.e., test case) corresponds to one or more CWE IDs, as multiple types of vulnerabilities could be identified in a program. We identify three major CFR vulnerability categories (i.e., CWE-691, CWE-840 and CWE-438) in the SARD. Thus, we have implemented a crawler to harvest all the available programs related to the following three CFR vulnerabilities from SARD.

- **Insufficient Control Flow Management Vulnerability (ICFMDS, CWE-691):** The code does not sufficiently manage its control flow during execution, creating conditions in which the control flow can be modified/referred in unexpected ways. There are 80 CWE-IDs belonging to this category according to CWE™ [9].
- **Business Logic Errors (BLEDS, CWE-840):** Vulnerabilities in this category allow attackers to manipulate the business logic errors of an application, which can be devastating to the entire application. However, many business logic errors can exhibit patterns that are similar to well-understood implementation and design weaknesses. 73 CWE-IDs belongs to this category.
- **Behavioral Problems (BHPDS, CWE-438):** Vulnerabilities in this category are related to unexpected behaviors from code that an application uses, such as incorrect control-flow execution orders. There are 29 CWE-IDs belonging to this category.

We conservatively label a sample code as '1' (i.e. vulnerable) if the code contains at least one vulnerable statement and '0' otherwise. Note that, most of programs in the SARD dataset were considered to be vulnerable. At last, our constructed benchmark contains 8,368 programs, including 8,343 vulnerable ones and 25 correct one, as shown in Table II. Note that, as our detection performs on the method granularity (CFG of each method), i.e., pinpointing whether a CFG of a method is vulnerable or not, thus we further analyze each program sample to label the vulnerable/accurate method, and use the CFG related features to train our model. **We have labelled 8,879 vulnerable method and 49,560 methods without any vulnerabilities, as shown in Table II**. Note that we also labelled the samples at the code-gadget granularity. Note that a code gadget is assembled by program slices [32], representing the statements of a program (i.e., lines of code) that are semantically related to an argument of a library/API function call [22], which will be used in our evaluation (cf. Section IV-C). Comparing with existing studies, we believe the benchmark we crafted is large enough to perform evaluation.

TABLE II
DISTRIBUTION OF LABELLED SAMPLES AT DIFFERENT GRANULARITIES.

Vul Category	granularity	vulnerable	correct	total
ICFMDS	program	624	13	637
	method	**850**	**4,563**	**5,413**
	code-gadget	80	224	304
BLEDS	program	3,859	6	3,865
	method	**4,014**	**22,497**	**26,511**
	code-gadget	871	1,980	2,851
BHPDS	program	3,860	6	3,866
	method	**4,015**	**22,500**	**26,515**
	code-gadget	869	1,982	2,851
TOTAL	program	8,343	25	8,368
	method	**8,879**	**49,560**	**58,439**
	code-gadget	1,820	4,186	6,006

B. Training the neural network

For the constructed benchmark, we randomly select 80% of the samples to train the neural network. We first introduce how we adapt the CFG samples to the existing neural network framework. Then we describe the details to implement the GCN model in this paper.

Data handling of graphs. A graph is used to model pairwise relations (edges) between objects (nodes). A single

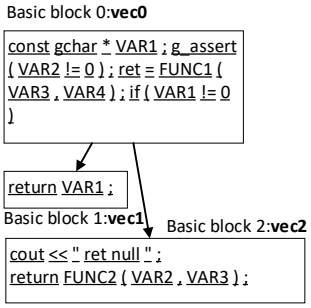


Fig. 8. Data handling of graphs. The CFG can be represented as $Data(x = [vec0, vec1, vec2], edge\_index = [[0, 0], [1, 2]], y = [0])$

graph in `PyTorch Geometric` [12] is described by an instance of `torch_geometric.data.Data`, which holds the following attributes by default:

- `data.x`: Node feature matrix with shape [num_nodes, num_node_features]
- `data.edge_index`: Graph connectivity in COO format with shape [2, num_edges] and type torch.long
- `data.y`: Target to train against (may have arbitrary shape)

Fig. 8 shows an example of the data handling process of the CFG. As can be seen from the figure, $vec0$ to $vec2$ are the vector representations of the two basic blocks. After getting the vector representations of the two basic blocks, we can form the $data.x$ matrix. The other two matrices can be easily obtained from the CFG's structure.

Implementing the GCN. We choose GCNConv [17] as the convolutional layer of our model. It is a scalable approach for semi-supervised learning on graph-structured data that is based on an efficient variant of convolutional neural networks which operate directly on graphs. We run experiments on a machine with NVIDIA GeForce GTX 1080 GPU and Intel Xeon E5-1620 CPU operating at 3.50GHz. The neural network is trained in a batch-wise fashion and the batch size is set to 64. We have implemented the graph convolutional neural network using `PyTorch Geometric` [12]. We adopt a 10-fold cross validation to train the neural network. We used grid search to perform hyper parameter tuning in order to determine the optimal values for a given model. The dimension of the vector representation of each basic block is set to 128. The dropout is set to 0.5 and the number of epochs is set to 10. The minibatch stochastic gradient descent together with ADAM [16] is used for training with the learning rate of 0.001. We used three convolutional layers on the datasets. The other parameters of our neural network were tuned in a standard method. All network weights and biases were randomly initialized using the default Torch initialization.

C. Evaluation Metrics

We have applied six widely used metrics including accuracy (ACC), false positive rate (FPR), false negative rate (FNR), true positive rate (TPR), F1-measure (F1) and AUC to evaluate the performance of vulnerability detection [25].

Accuracy means the correctness of all detected samples. *FPR* means the proportion of false-positive samples in the total samples that are correct. *FNR* means the proportion of false-negative samples in the total samples that are vulnerable. *TPR* means the proportion of true-positive samples in the total samples that are vulnerable. *F1-measure* means the overall effectiveness considering both precision and false negative rate. *AUC* means the area under the receiver operating characteristic (ROC) curve. According to previous research, AUC and F1-measure has been proved as a better and more statistically consistent criterion than the accuracy [23], especially for imbalanced data.

IV. Results and Analysis

To evaluate the performance of VGDetector, we have compared it with two well-known static vulnerability detection frameworks (Flawfinder [4] and RATS [6]), and two state-of-the-art machine-learning-based approaches (Vuldeepecker [22] and token-based embedding [33]). Table III shows the overall results on the benchmark in terms of all the aforementioned evaluation metrics. In general, VGDetector performs the best out of all the studied frameworks, and the average detection accuracy is over 91% for all the three CFR vulnerabilities.

A. VGDetector *VS. Traditional Detection Tools*

We consider two open source tools Flawfinder [4] and RATS [6], as the baseline. These two tools are open-sourced and widely used by the community. These frameworks detect vulnerabilities based on a list of pre-defined anti-patterns, and they mainly target low-level bugs including buffer overflow risks, race conditions, format string problems, and so on. The detection process is quite simple, by processing the source code and matching the summarized anti-patterns directly.

To enforce the fair comparison, we have applied these two frameworks to detect vulnerabilities in our labeled benchmark on the method granularity, i.e., predicting a method is vulnerable if it contains at least one vulnerable statement flagged by these tools. As shown in Table IV, VGDetector outperforms the referred tools with regard to all the evaluation metrics. The false negative rates of both tools are over 50% in all the three vulnerability categories we considered. Surprisingly, for the ICFMDS vulnerability, over 72% of the samples reported by the tool RATS are false negatives. This result suggests that these tools are almost cannot be adopted in the real-world software systems, as human experts need to manually re-check almost all the reported methods due to high FPs and FNs.

The main reason leading to the poor performance of the traditional bug detection tools is that, the high-level CFR bugs are often triggered due to bad yet complicated programming practices, which have no specific bug patterns, thus posing a great challenges for the pattern-based detection tools. By manually examining the rules defined by human experts that embedded in the tools, we found the rules are quite simple and the number of rules is quite limited. Take FlawFinder [4] as an example, the size of its rule-set is 223, which means that it only relies on the 223 code patterns to detect a number

TABLE III
COMPARING VGDETECTOR WITH STATE-OF-THE-ART VULNERABILITY DETECTION APPROACHES. THE BEST RESULT OF EACH METRIC IS SHOWN IN BOLD.

TABLE IV
COMPARING WITH TRADITIONAL TOOLS

Category	Method	TPR	FPR	FNR	ACC	F1	AUC
ICFMDS	RATS	0.271	0.146	0.729	0.825	0.134	N/A
	Flawfinder	0.371	0.214	0.629	0.764	0.137	N/A
	VGDETECTOR	**0.846**	**0.069**	**0.153**	**0.918**	**0.766**	**0.955**
BLEDS	RATS	0.464	0.192	0.536	0.759	0.350	N/A
	Flawfinder	0.472	0.363	0.528	0.614	0.255	N/A
	VGDETECTOR	**0.894**	**0.066**	**0.106**	**0.926**	**0.791**	**0.977**
BHPDS	RATS	0.464	0.193	0.536	0.759	0.350	N/A
	Flawfinder	0.472	0.363	0.528	0.614	0.255	N/A
	VGDETECTOR	**0.899**	**0.066**	**0.101**	**0.929**	**0.792**	**0.975**

TABLE V
COMPARING WITH VULDEEPECKER AND TOKEN-BASED METHOD

Category	Method	TPR	FPR	FNR	ACC	F1	AUC
ICFMDS	Vuldeepecker	0.404	0.065	0.596	0.793	0.494	0.862
	Token-based	0.600	**0.011**	0.399	**0.928**	0.722	0.926
	VGDETECTOR	**0.846**	0.069	**0.154**	0.918	**0.766**	**0.955**
BLEDS	Vuldeepecker	0.717	0.187	0.282	0.784	0.667	0.866
	Token-based	0.572	**0.011**	0.428	0.926	0.700	0.933
	VGDETECTOR	**0.894**	0.066	**0.106**	**0.926**	**0.791**	**0.977**
BHPDS	Vuldeepecker	0.697	0.181	0.303	0.782	0.657	0.874
	Token-based	0.609	**0.015**	0.391	0.928	0.719	0.933
	VGDETECTOR	**0.899**	0.066	**0.101**	**0.929**	**0.792**	**0.975**

of different vulnerabilities. It is known to us that, the real-world vulnerabilities are far more complicated than the simple rules defined by the detection tools, especially for the high-level CFR bugs. As time goes by, diverse kinds of high-level vulnerabilities will occur, which will great limit the usage scenarios of the traditional rule-based detection tools, as they are purely relying on human experts to craft the sophisticated detecting rules.

To conclude, our experiment results suggest that the traditional rule-based detection method is not applicable to detecting high-level CFR vulnerabilities, while VGDETECTOR, which embeds high-level control-flow information via deep-learning, is effective in pinpointing CFR vulnerabilities in a general manner without the knowledge of any pre-defined anti-patterns.

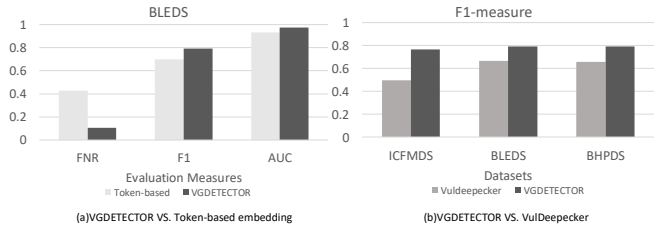


Fig. 9. Comparing VGDETECTOR with token-based embedding approach and Vuldeepecker. Fig (a) compares token-based embedding with VGDETECTOR on BLEDS vulnerability. Fig (b) compares VGDETECTOR with Vuldeepecker on the three kinds of vulnerabilities using F1-measure.

B. VGDETECTOR *VS. Token-based Embedding*

Token-based embedding approach [33] was proposed to detect vulnerabilities by representing the source code as sequential tokens. It first generates a token sequence for each method and then embeds raw-text information via deep-learning. It claims to support both low-level bugs and high-level ones, as bugs are nowhere else except in the code raw-text.

As shown in Table V, VGDETECTOR outperforms the token-based embedding approach considering all the three vulnerabilities and all the evaluation metrics. Taking the Business Logic Error bug (BLEDS) as an example, we further illustrate the result in Fig 9(b) in a more intuitive manner. It is interesting to see that, the false positive rate of the token-based approach is roughly 42%, while the percentage of our approach is 10%, only a quarter of referred approach. With regard to the result of F1-measure, VGDETECTOR is roughly 10% higher than the token-based approach.

This result suggests that, representing the source code as raw-text is far from enough to detect vulnerabilities, as the semantic information is lost during the process, which is vital for predicting vulnerabilities. As a contrast, VGDETECTOR uses a more precise code abstraction (i.e. CFG) as the representation of source code. Figuratively speaking, token-based embedding only considers the methods of source code as single-big code blocks (i.e. sequential tokens), while VGDETECTOR splits them up and add connections between smaller code blocks (i.e. basic blocks). In general, the real-world programs are complex, and the CFG used in our approach can better reflect the execution logic of the code while the token sequence only considers the code as a plain text, missing the execution logic and the information of relationships between basic blocks. Taking "if-else" as a simple example, the 'if' scope and 'else' scope should be considered as the same level while token-based approach represents them in a sequence manner, without considering the branch information.

C. VGDETECTOR *VS. VulDeepecker*

Vuldeepecker [22] is a state-of-the-art approach that detects vulnerabilities in source code, which relies on data-flow dependency information to represent the source code. It first locates bug-related APIs in the source code and then extracts program slices corresponding to the parameters of these APIs, thus generating the code gadgets by assembling api-related program slices [22] to enforce accurate bug detection. It uses code gadget as the granularity to pinpoint bugs and the number of generated code gadgets depends on the number of APIs existed in the source code. As shown in Table II, we have extract the code gadgets in our benchmark to evaluate the performance of Vuldeepecker.

As illustrated in Table V, VGDETECTOR outperforms Vuldeepecker greatly, as Vuldeepecker reports a large number of false positives and false negatives. For example, considering the ICFMDS vulnerability, the false negative rate of Vuldeepecker has achieved 59.6%, which suggested that over 40% of the vulnerable samples were overlooked by Vuldeepecker. Even the token-based embedding approach performs better than Vuldeepecker in our dataset. This result suggested that although

Vuldeepecker claims to perform well on low-level bugs such as buffer over flow, it cannot handle CFR bugs perfectly.

The main reason is that, CFR vulnerabilities are mostly control-flow-related, while Vuldeepecker only considers data flow dependency, without considering the control flow such as branch conditions. Fig 10 shows an example of Insufficient control flow management error (CWE-691). Following the approach presented in Vuldeepecker, we extract the code gadgets from the source code. As shown in Fig 10, there is no difference between the code gadgets of the vulnerable code and accurate one. They are the same code gadget (i.e. lines of code). However, the CFGs of vulnerable code and the correct one are quite different, considering the relationships between basic blocks. Therefore, VGDETECTOR is more effective in detecting such kinds of vulnerabilities.

Furthermore, as mentioned in VulDeepecker [22], because of the difficulty of locating the key-point (i.e. APIs) of vulnerability in some cases, it can only extract a limited number of code gadgets from the labelled samples, which can not uncover the vulnerabilities that are not related to key APIs. By contrast, VGDETECTOR extracts all the methods in the source code, so theoretically it can detect all types of vulnerabilities as long as the dataset is large enough to learn from practice.

D. False Positives and False Negatives of VGDETECTOR

Although our extensive experiments suggest that VGDETECTOR outperforms all the traditional tools and machine-learning based frameworks, VGDETECTOR still reports false positives (roughly 6%) and false negatives (10% to 15%) across different vulnerability categories.

By manually examining some exceptional cases, we found that the following reasons may lead to the outlier. First, the pre-processing procedure is not perfect, as we manually summarized a list of rules to unify the codes that preserves the same semantics, which is quite possible to be incomplete. This may have bad influence on the result. Moreover, VGDETECTOR directly learns from source code using a general representation (i.e. CFG) but with "noise" in it. As some tokens and statements are not related to bug detection, we need to identify and eliminate such information to perform better feature abstraction. However, it is non-trivial for us to filter such information in practice. Besides, as the goal of this work is to characterize control-flow-related bugs, we mainly focus on control-flow semantics during the abstraction and embedding process, without performing the data-flow analysis (e.g., reaching definition analysis). As some vulnerabilities are both control-flow and data-flow related, it is a major limitation for us to detect such vulnerabilities.

V. THREATS TO VALIDITY

This section discuss several limitations of our approach to give insights for future research.

First, we mainly focus on dealing with control-flow-related vulnerabilities in this paper. For traditional type of memory errors such as buffer overflows and null pointers are out of scope of this work. Incorporating data-flow with control-flow information to enhance our vulnerability detection can also be an interesting research direction. Second, our experiments focus on dealing with vulnerabilities that happen in C/C++ programs. However, it is easy to extend our framework to support other programming languages, e.g., Java and Python. Third, the current implementation of VGDETECTOR is limited to the graph convolution network. Exploring other type of neural networks is also an interesting research direction. Finally, it is also interesting to investigate how to cope with varying lengths of the vector representation of a basic block without losing the information caused by the truncation.

VI. RELATED WORK

There are a wide variety of analysis tools and researches in the area of software vulnerability detection. We discuss two main dimensions of static analysis (closely related to this paper) for detecting software vulnerabilities, including traditional approaches and machine learning-based approaches.

There are plenty of traditional static program analyzers for analyzing large software systems (e.g. SVF [19], Clang [34], Coverity [3], Fortify [5], Flawfinder [4], ITS4 [30], RATS [6], Checkmarx [2]). They have been show their effectiveness in detecting well-defined low-level bugs, such as memory errors. However, they often suffer from a large number of false positives and/or false negatives in detecting high-level vulnerabilities, such as CFR bugs. These traditional static analyzers heavily rely on conventional static analysis theories (e.g., data-flow, abstract interpretation and taint analysis), which are difficulies in understanding and identifying CFR vulnerabilities manifested as complicated high-level bug patterns.

Another branch for statically detecting vulnerabilities is to employ code similarity analysis (e.g. detecting vulnerabilities due to code clones in the forms of software patches [14], common modifications [15], duplicated product lines [20], code retrieval [31] and regression bugs [26]). Code similarity analysis normally extracts each code fragment into an abstract representation and then computes the similarity between pairs of abstractions. These approaches, however, require human experts to define features in order to apply appropriate code similarity algorithms for different types of vulnerabilities [27]. In contrast, our approach can detect vulnerabilities in a fully automatic manner using graph embedding.

Recently, there are also several static bug detection approaches to detecting well-defined low-level vulnerabilities using machine learning techniques. Neuhaus et al. [24] apply support vector machines to identify bugs in Red hat packages. Shin et al. [28] selectively apply unsound static analysis to perform taint and interval analyses to reduce false alarms while retaining true alarms. Yan et al. [35] perform machine-learning-guided type state analysis for detecting use-after-frees. Vuldeepecker [22] applies code embedding using data-flow information of a program for detecting resource management errors and buffer overflows. Compared with these approaches that mainly focus on detecting low-level bugs, our approach focuses on detecting high-level control-flow-related vulnerabilities with low false positive and negative rates.

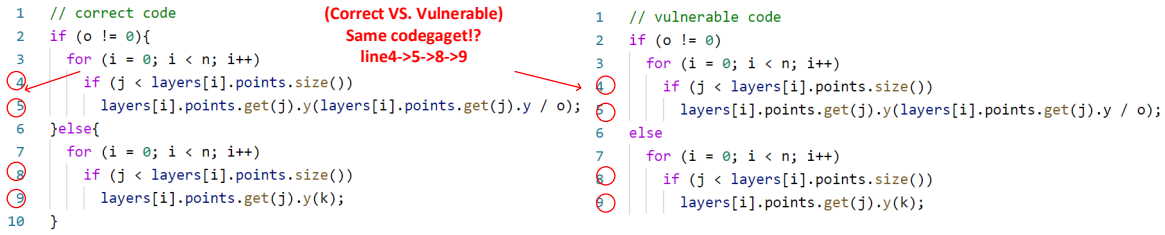


Fig. 10. An example of insufficient control flow management vulnerability.

VII. Conclusion

This paper presents VGDetector, a new deep-learning-based graph embedding approach to accurate detection of control-flow-related vulnerabilities. Our approach makes a new attempt by applying a recent graph convolutional network to embed code fragments in a compact and low-dimensional representation that preserves high-level control-flow information of a vulnerable program. Our experiments show that VGDetector outperforms several popular traditional and machine-learning-based static detectors. Our observations have shed light on the promising direction of combining program analysis with deep learning to address the general static analysis challenges.

VIII. Acknowledge

This work is supported by the National Key Research and Development Program of China (grant No.2018YFB0803603), the National Natural Science Foundation of China (grants No.61702045 and No.61897069), and the Australian Research Grant DE170101081. Prof. Haoyu Wang and Dr. Li Yi are corresponding authors.

References

[1] Antlr. https://www.antlr.org/.
[2] Checkmarx. https://www.checkmarx.com/.
[3] Coverity. https://scan.coverity.com/.
[4] Flawfinder. https://dwheeler.com/flawfinder/.
[5] HP Fortify. https://www.hpfod.com/.
[6] RATS. https://code.google.com/archive/p/rough-auditing-tool-for-security/.
[7] Tscancode. https://github.com/Tencent/TscanCode.
[8] Software Assurance Reference Dataset, 2017. https://samate.nist.gov/SARD/index.php.
[9] common weakness enumeration, 2019. https://cwe.mitre.org/index.html.
[10] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. In *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, Nov 2014.
[11] Shadi A. Aljawarneh, Ali Alawneh, and Reem Jaradat. Cloud security engineering: Early stages of sdlc. *Future Generation Computer Systems*, 74:385 – 392, 2017.
[12] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
[13] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 519–529, Piscataway, NJ, USA, 2017. IEEE Press.
[14] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62, May 2012.
[15] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.
[16] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec 2014.
[17] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv e-prints*, page arXiv:1609.02907, Sep 2016.
[18] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1188–1196. JMLR.org, 2014.
[19] Yuxiang Lei and Yulei Sui. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *SAS '19*, 2019.
[20] J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *Proceedings of ICSE '12*, pages 310–320, 2012.
[21] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: An automated vulnerability detection system based on code similarity analysis. In *Proceedings of ACSAC '16*, pages 201–213, 2016.
[22] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *The Network and Distributed System Security Symposium (NDSS)*, 2018.
[23] Charles X. Ling, Jin Huang, and Harry Zhang. Auc: A statistically consistent and more discriminating measure than accuracy. In *Proceedings of IJCAI'03*, pages 519–524, 2003.
[24] Stephan Neuhaus and Thomas Zimmermann. The beauty and the beast: Vulnerabilities in red hat's packages. In *In Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC*, 2009.
[25] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Comput. Surv.*, 49(4):62:1–62:35, December 2016.
[26] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 447–456, 2010.
[27] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199, 2013.
[28] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Software Eng.*, 37:772–787, 11 2011.
[29] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266, 2016.
[30] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267, Dec 2000.
[31] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. Multi-modal attention network learning for semantic source code retrieval. In *ASE '19*, 2019.
[32] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
[33] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of MSR '15*, pages 334–345, 2015.
[34] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of c programs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, 2010.
[35] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *ACSAC '17*, pages 42–54. ACM, 2017.