Combining Graph-based Learning with Automated Data Collection for Code Vulnerability Detection

Huanting Wang[1,*], Guixin Ye[1,*], Zhanyong Tang[1,✉], Shin Hwei Tan[2]
Songfang Huang[3], Dingyi Fang[1], Yansong Feng[4], Lizhong Bian[5], and Zheng Wang[6,✉]
1. Northwest University, China; 2. Southern University of Science and Technology, China;
3. Alibaba DAMO Academy; 4. Peking University, China;
5. Alipay (Hangzhou) Information & Technology Co., Ltd.; 6. University of Leeds, U. K.

***Abstract*—This paper presents FUNDED[1], a novel learning framework for building vulnerability detection models. FUNDED leverages the advances in graph neural networks (GNNs) to develop a novel graph-based learning method to capture and reason about the program's control, data, and call dependencies. Unlike prior work that treats the program as a sequential sequence or an untyped graph, FUNDED learns and operates on a graph representation of the program source code, in which individual statements are connected to other statements through relational edges. By capturing the program syntax, semantics and flows, FUNDED finds better code representation for the downstream software vulnerability detection task. To provide sufficient training data to build an effective deep learning model, we combine probabilistic learning and statistical assessments to automatically gather high-quality training samples from open-source projects. This provides many real-life vulnerable code training samples to complement the limited vulnerable code samples available in standard vulnerability databases.**

We apply FUNDED to identify software vulnerabilities at the function level from program source code. We evaluate FUNDED on large real-world datasets with programs written in C, Java, Swift and Php, and compare it against six state-of-the-art code vulnerability detection models. Experimental results show that FUNDED significantly outperforms alternative approaches across evaluation settings.

***Index Terms*—Software Vulnerability, Code Vulnerability Detection, Deep Learning, Deep Graph Neural Networks**

I. INTRODUCTION

Software vulnerabilities are responsible for many system attacks [1] and data breach incidents [2]. Machine learning is a viable means for constructing tools and models to identify common software vulnerabilities. It works by first learning, from training samples, the latent patterns indicative of vulnerable programs. A machine-learned model can then be applied to new software projects to identify potentially vulnerable code that exhibits similar patterns as those vulnerable samples seen in the training data. There is now ample evidence showing that machine learning techniques can exceed expert-crafted rules [3] for detecting common code vulnerabilities or bugs.

Recent studies have leveraged deep learning (DL) to reason about program structures to identify potential software vulnerabilities at the source code [4, 5, 6, 3, 7]. Compared to classical machine learning techniques, DL has the advantage of not requiring expert involvement to tune representations for program structures manually; instead, it automatically captures and determines them from training samples.

Existing DL-based approaches for program modeling typically use recurrent neural networks (RNNs) such as the Long Short-Term Memory (LSTM) or a variant of it [5, 6, 8, 3, 7]. These approaches work by treating source code and its corresponding program structure, such as the abstract syntax tree (AST), as a sequence of tokens. However, LSTM is designed for sequential sequences [9] and is ill-suited for modeling the well-structured control and data flows of programs. As a result, prior LSTM-based methods only capture the shallow, surface structure of the source code text and fail to capitalize on the rich and well-defined semantics of the program structure. As shown in our evaluation, existing LSTM-like approaches often give poor accuracy, either missing vulnerabilities or giving overwhelmingly false-positive results.

To better model the complex code structures - which were traditionally represented as graph structures in compilers for code analysis [10] - we need an approach that could directly operate on and learn from the graph representation of the code. Doing so will allow the learning framework to preserve and reason about much of the control and data flow information for capturing the essential code structures for many software vulnerabilities. For example, to detect the *use-after-free* vulnerability, we need to know where and when a buffer is allocated and deallocated across multiple execution paths.

We introduce FUNDED, a better approach for modeling code structures. FUNDED operates on graph representations of the program source code with the capability to learn and aggregate multiple code relationships. It achieves this by leveraging the recently proposed gated graph neural networks (GGNNs) [11]. By directly operating on a graph representation, the graph neural networks (GNNs) have shown astounding successes in social networks [12], and knowledge graphs [13] and even compiled binaries [14]. While GNN provides a good starting point, applying it to develop a practical and efficient framework for software vulnerability detection is not trivial. As a standard GNN operates on a *single* graph representation with *untyped* edges, it cannot distinguish between the control and data flow information. However, such information is essential for capturing vulnerable and buggy code patterns. As demonstrated by our evaluation, when ignoring the different code relationships, a recent work that uses a vanilla GNN [15] gives marginal improvement compared to the LSTM

This work was supported in part by the National Natural Science Foundation of China (NSFC) under grant agreements 61972314, 61672427 and 61872294, The International Cooperation Projects of Shaanxi Province under grant agreements 2019KW-009 and 2020KWZ-013, an Ant Financial Science funded project and an Alibaba Innovative Research Program.

✉ Corresponding authors: Zhanyong Tang (zytang@nwu.edu.cn) and Zheng Wang (z.wang5@leeds.ac.uk))

*Huanting Wang and Guixin Ye are the co-first authors.

[1]FUNDED = **F**low-sensitive v**U**l**N**erability co**DE** **D**etection.

alternatives for code vulnerability detection.

FUNDED extends the GNN's capability to distinguish and model multiple code relationships (including data, control, operation order, and operand values). This is achieved by first encoding different code relationships in different relation graphs, and then using learnable, relation-specific functions to propagate and aggregate information across relation graphs. By representing the input program as multiple relation graphs with explicit control and data flows or syntactic information, our new graph model captures richer intra-program relations than prior GNN-based approaches [15]. This richer set of relationships improves the model's ability in learning useful program representation, leading to better performance of the downstream code vulnerability detection task. As we will show later, by employing the GGNN to model and distinguish the rich code relationships, our approach significantly outperforms alternative graph-based methods.

While our novel GNN extension provides a potentially powerful capability for learning code representation, its potential can only be unlocked with sufficient training data. Typical DL algorithms require up to millions of examples to learn an efficient model [16], but the scarcity of real-life vulnerable training samples is a common problem [3]. The lack of training data limits the quality of machine-learned detection models, as they have very sparse training data for typical high-dimensional program space. Some prior approaches solve this problem using program generation for compiler testing [17]. However, synthetic programs have two significantly drawbacks. They are biased by the grammars, templates, or models used to generate the programs, and may not reflect the diverse and evolving patterns of real-life programs. Hence, models learned over synthetic data are hard to generalize.

Our solution for addressing the scarcity of vulnerability trainining data is to utilize the wealthy historical information in open-source projects. We achieve this by using an *off-line* trained model to predict which code commit is used to patch a code vulnerability. The *vulnerability-relevant commit* - i.e., a code revision that provides a patch for a code vulnerability - is then used to locate a vulnerable source code snippet from the version before the patch commit.

Translating this high-level idea to build a practical data collection system is, however, not trivial. Collecting a large number of high-quality vulnerable code samples are challenging because we need to exclude commits that are irrelevant to software vulnerabilities (e.g., code commits for enhancing performance or functionalities but not repairing a vulnerability). Failure to exclude such benign code snippets in our training samples will confuse the machine learning algorithm. Meanwhile, it is impractical to ask developers to manually check all the training samples due to the large volume of data collected. Thus, we must find a better measure to ensure the quality of the collected data and only ask for developer intervention when necessary.

Prior work on vulnerable training sample collection [18, 19, 20] is characterized by a one-size-fits-all assumption. They use a single monolithic model for locating vulnerability-relevant commits. These approaches often fail to examine whether the model fits the current inputs or whether another model would perform better. However, machine learning is well known to be brittle to uncertainties. When facing an situation that is not seen before, machine learning techniques often produce an answer with a high probability. The high probability in the uncertain situations often lead to poor prediction results – in our case, this will introduce noise into training data and deteriorate the quality of the learned model.

To ensure the quality of the training data, we take a different approach by adopting a "mixture of experts" scheme [21] to collect training data from open-source projects. Our approach reduces data noise by employing multiple predictive models (referred to as *experts*) and only using predictions (or expert recommendations) that we have high confidence on the model's output. To evaluate the confidence (or certainty) of each recommendation, we apply Conformal Prediction [22] to measure the statistically valid confidence for the predictions given by individual models. In this way, we use only recommendations with high confidence. Given this ability to measure the confidence of predictions, we only ask developers to inspect low-confident predictions to provide the ground-truth, which then serves as additional training data to improve the data collection model over time. We show that our mixture-of-expert approach improves the quality of the collected training data, leading to a better-performing vulnerability detection model. This also provides a scheme to gradually and continuously update the data collection model with minimum developer involvement.

We demonstrate the benefits of FUNDED by applying it to detect function-level vulnerabilities from program source code[2]. We thoroughly evaluate FUNDED on large real-life datasets of code commit history and vulnerable programs written in C, Java, Php and Swift. We compare FUNDED against six state-of-the-art (SOTA) learning-based detection methods for software bugs or vulnerabilities [4, 5, 16, 6, 3, 15], and five SOTA methods for automatic vulnerable code sample collection [18, 19, 23, 20, 24]. Experimental results show that FUNDED consistently outperforms competing methods across evaluation settings, by discovering more code vulnerabilities with a lower false-positive rate.

Contributions. This paper is the first to:

- show how a multi-relational, gated graph neural network can be developed for vulnerability detection (Sec. IV);
- combine probabilistic learning and statistical assessment to develop a "mixture-of-experts" approach to address the shortage of vulnerable training code samples (Sec. V);
- exploit transfer learning to port vulnerability detection models across programming languages (Sec. VII-D);

II. BACKGROUND

A. Problem Scope

FUNDED is a general learning framework for code vulnerability detection. In this work, we apply FUNDED to identify vulnerabilities from the source code. FUNDED predicts if a given function or method contains a potential vulnerability and of what type. Here, the target function may invoke standard library calls and user-defined functions. Note that our intention

[2]Code and data available at: https://github.com/HuantWang/FUNDED_NISL

```
1 attr_value = (char*)malloc(attr_len + 1);
2 ...
3 else if(!strcmp(attr_name, "dateadded"))
4 {
5   ae->date_added = atoi(attr_value);
6   free(attr_value);
7 }
8 else
9   free(attr_value);
```

Fig. 1. Benign code sample from GitHub. VULDEEPECKER, μVULDEEPECKER and LIN *et al.* all misclassify the code containing a "*double-free*" vulnerability for buffer `attr_value`.

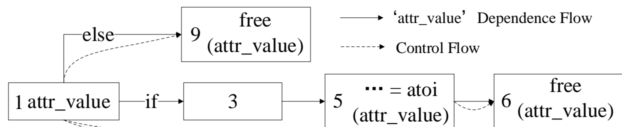


Fig. 2. Control and data flow for buffer `attr_value` in Figure 1. The line number is given in each rectangle box.

is not to uncover a new type of vulnerabilities. Instead, we want to detect if a new, unseen piece of code contains a vulnerable code pattern that is similar to one seen in the training dataset. Therefore, FUNDED is useful for detecting common, repeatedly occurred software vulnerabilities (or bugs). To this end, our work focuses on detecting common vulnerabilities (or weaknesses) defined in the common weakness enumeration (CWE) database, and is not concerned about non-vulnerable bugs like performance issues.

B. The Need for Flow-sensitive Methods

To show the need for modeling the control and data flows, consider the benign code example given in Figure 1. VULDEEPECKER [5], μVULDEEPECKER [6] and LIN *et al.* [3] are SOTA vulnerability detection models. They build upon bidirectional LSTM (BiLSTM) [25] - a sequence deep-learning model. For this example, they all incorrectly classify the code containing a *double-free* vulnerability. The root cause for this *false positive* is that a sequence model has to linearize and treat the code structure as a sequential sequence of tokens, which omits the control flow divergence (see Figure 2). Consequently, they regard dynamic buffer `attr_value` (line 9) to be deallocated again after it being freed at line 6.

For this example, we want to capture the control and data flow of the target program by using a flow-sensitive decision model. If we can do that, we can then infer that the buffer `attr_value` at line 9 is deallocated in a different execution path and hence will not lead to a *double-free* vulnerability. For more general cases, we need to capture syntactic information (when trading sequential representations for graphs), as well as the control and data flows or any other code relationships that may be essential for the downstream processing task. FUNDED is designed to offer such capabilities by extending the recently proposed GNN architecture.

III. OVERVIEW OF OUR APPROACH

FUNDED consists of two key components. The first is a GNN-based model to identify potential software vulnerabilities at the source code level. The second is an automatic framework for collecting vulnerable code samples from open-source repositories (i.e., GitHub in this work) to provide additional training data for learning the vulnerability detection model.

TABLE I
GITHUB COMMIT EXAMPLES

Code revisions	C1: Vulnerability-relevant commit	C2: Vulnerab. irrelevant commit
Message	Add NULL check to avoid null pointer access.	Check err when partial == NULL is meaningless because partial == NULL means getting branch successfully without error.
Patch	*4 addition lines , 2 deletion lines* -sap_ctx->csa_reason = reason; +if (sap_ctx) +sap_ctx->csa_reason = reason; -hdd_ap_ctx->... +if (hdd_ap_ctx -> sap_context) +hdd_ap_ctx->...	*3 addition lines, 2 deletion lines* -if (err) -goto cleanup; +if (err) +mutex_unlock(ei->... +goto cleanup;

A. Software Vulnerability Detection

Our vulnerability detection model builds on a new GNN proposed in this work. The model takes as input source code of the target function. Next, it constructs a *program graph* by combining information extracted from the abstract syntax tree (AST), and the program control and dependence graph (PCDG). The program structures are presented as directed graphs, where statements, identifiers, and immediate values are graph nodes, and a direct relationship (e.g., parent-child, data or control flow, etc.) between two nodes is recorded as an edge. As there may exist multiple relationships (or edges) among a pair of nodes, we use a relation graph to record each type of relationships (see Sec.IV-C2). The node connectivity of a relation graph is encoded as a program graph matrix. Our GNN takes in the program matrices and initial node representations to learn code representations called embeddings that are represented as a vector of numerical values. The code embeddings are passed to a downstream neural network to make a prediction.

The detection model is trained *offline* using training datasets from both standard software vulnerability databases like CVE and SARD, and examples gathered from open-source repositories. The trained model can then applies to any "*new*, *unseen*" programs. Unlike prior work [4, 5, 3, 16, 6, 15], our multi-relational GNN can better capture multiple code relationships, leading to significantly more accurate detection results. We describe the vulnerability detection model in Sec. IV.

B. Training Data Collection

To gather vulnerable code samples from open-source repositories like GitHub, we develop an automatic data collection framework. This framework aims to provide real-world vulnerable code samples, complementary to those available from standard vulnerable databases like the common vulnerabilities and exposures (CVE) and the software assurance reference dataset (SARD). Our framework uses a set of predictive models or *experts*, each independently predicts whether a code commit provides a patch for a code vulnerability in the previous version of a software project. By identifying vulnerability-relevant code commits, we can examine the changes brought

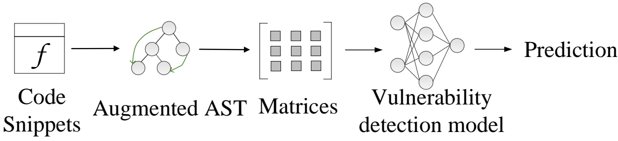


Fig. 3. Code vulnerability detection. Our detection model takes as input an AST and CDFG of the target code snippet at the function level.

by the patch to locate which code segments of the previous version are likely to lead to a vulnerability. The identified code segment (a function or method in this work) is then used as a vulnerable code[3] training example. This automatic data collection framework enables us to build a large training dataset of real-life programs.

Consider the two commits in Table I from a customized Linux kernel hosted on GitHub. The first commit (C1) fixes a NULL pointer vulnerability and the second commit (C2) fixes a performance issue but not vulnerability. For the detecting vulnerable code, code extracted from the second commit should be excluded from the vulnerable training samples. However, existing approaches (VCCFinder [18] and Zhou *et al.* [23]) may incorrectly label the second revision in Table I as vulnerability-relevant commit because the commit message contains keywords "check" and "NULL". To avoid such mistakes, we apply Conformal Prediction (CP) to quantify the confidence (or credential) of each expert model's prediction (or recommendation) for a commit and only consider predictions when we trust the model's outcome. The use of CP helps us to improve the quality of the collected data. We describe this training data collection framework in Sec. V.

Roadmap. In the following two sections, we first describe our GGNN-based code vulnerability detection model in Sec. IV and then present our training data collection model in Sec. V.

IV. Detecting Code Vulnerabilities

In this section, we describe our GGNN-based code vulnerability detection model. We start by giving an overview of our model in Sec. IV-A. We then move to explain the model structure in Sec. IV-B before describing how we organize the program structures as a graph input to the model in Sec. IV-C. This is followed by a detailed description of the training and learning process of our model in Sec. IV-D, Sec. IV-E and Sec. IV-F. Finally, we discuss the model interpretability issue in Sec. IV-G.

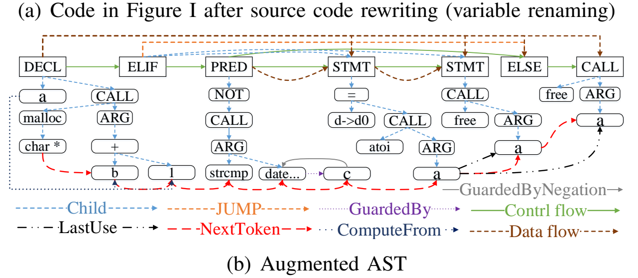
A. Overview of Our Detection Model

Figure 3 depicts the workflow of our detection model, which takes as input the source code of the target program (i.e., a function). We construct an AST of the code using a standard compiler parser. We extend the AST with additional control and data flows and sequential information like the token sequence. The extended AST are presented as directed multiple

[3] In this work, a *vulnerability-relevant code commit* = a code revision log that provides a fix for a vulnerability, while *vulnerable code* = a piece of code that contains a type of vulnerability defined in the CWE. We stress that identifying vulnerability-relevant code commits is fundamentally different from identifying vulnerabilities from source code as we can utilize additional information like commit messages and code changes between two commits to assist with the former task.

```
1  a = (char*)malloc(b+1);
2  ...
3  else if(!strcmp(c,"dateadded")) {
4    d->d0 = atoi(a);
5    free(a);
6  } else {
7    free(a);
8  }
```

(a) Code in Figure I after source code rewriting (variable renaming)



(b) Augmented AST

Fig. 4. Normalized code (a) and the extended AST (b).

graphs, where statements, code blocks or values are graph nodes, and a direct relationship (e.g., parent-child and other relationships between two nodes) is recorded as an edge. As there may exist multiple relationships among a pair of nodes, we use a relation graph to record each type of relationships (nine relationships in total). The node connectivity of a relation graph is encoded as an adjacency matrix.

B. GNN Model Structure

Building upon our recent work [26], we extend the gated graph neural network (GGNN) [27] to model multiple code relationships extracted from the source code. Our GGNN consists of four stacked embedding models based on the Gated Recurrent Unit (GRU) [28], so that it can incorporate higher degree neighborhoods across relation graphs. It takes in the adjacency matrices of relation graphs and initial node representations to learn a global embedding vector, which is then passed to a standard fully-connected network to make a classification using a softmax layer.

C. Graph Representations

1) Code preprocessing: As a preprocessing step, we use a compiler parser to rewrite the variable names using a consistent naming scheme. This step ensures that trivial semantic differences in programs such as the choice of variable names do not affect the choice of token embeddings (Sec. IV-D). Figures 4(a) shows the source rewriting applied to the example shown in Figure 1.

2) Program graph: Our program graph is constructed from the AST that contains *syntax nodes* (i.e., nonterminals in the language grammar, e.g., an AST node for an `if` statement or function declaration) and *syntax tokens* (terminals like identifier names and constant values). A standard AST has just only the `child` edge for encoding the parent-child relationships between two AST nodes. To capture additional syntax, data and control information, we add eight additional types of edges to the AST, following the methods described in [29]. We describe our additional edges as follows.

Data and control flows. We integrate the data and control paths extracted from the PCDG to the AST.

GuardedBy. We connect each AST token of a variable to the variable's enclosing guard expressions using a `GuardedBy` edge. For example, for the `if` statement in Figure 4(a), we add a `GuardedBy` edge from `d` and `free(a)` to the AST node corresponding to `!strcmp()`. This could be useful for determining the wrong order of operands [16].

Jump. We use a `Jump` edge to connect variables with control dependencies. The `GuardedBy` and `Jump` edges allow us to record the relationship of diverging control flows. Such a relationship is important for capturing control and data flow patterns for vulnerabilities like the "*double-free*" example given in Figure 1 and "*CWE-413: improper resource locking*".

ComputedFrom. For each assignment, $v = expr$, we connect v to all variable tokens occurring in expression, $expr$, using `ComputedFrom` edges. This edge captures where a variable or buffer is used and is useful for detecting vulnerabilities like "*NULL pointer dereference*".

NextToken. As the standard AST child-parent edge does not induce an order on children of a syntax node, we add `NextToken` edges to connect each syntax token to its successor. This is used to capture the order of opcode and operands for statements. Such information is useful for vulnerability types like"*CWE-404: improper resource shutdown or release*" because it captures the order of API uses and releases.

LastUse and LastLexicalUse. We connect all uses of the same variable using `LastUse` edges to capture the use of variables, where a special case is variables in `if` statement and we connect such type of variables using `LastLexicalUse` edges. For instance, for the `if` statement in Figure 4(a), we would link the occurrences of `c` in the loop head and where it is used. By recording when a variable or buffer is last used, this relationship helps in identifying vulnerabilities like "*double-free*". Figure 4(b) shows the augmented AST after processing the code given in Figure 4(a).

3) Relation graphs: We store the relationships of the augmented AST in separate relation graphs - one graph for each of the nine relationships described above. In this work, a relation graph is a directed graph, $\mathcal{G} =< \mathcal{V}, \mathcal{E} >$, that contains the AST nodes, $\mathcal{V}$, and edges $\mathcal{E}$, that indicate the existence of a given relationship between two nodes. We use an adjacency matrix to record the edge connections of each relation graph. For each edge, we also add a respective *backward* edge (by transposing the adjacency matrix), doubling the number of edges and edge types. These backward edges help with propagating information across relation graphs.

D. Graph Node Representations

We map every program graph's nodes (e.g., `Stmt`) and tokens to an *embedding* vector of numerical values using a word2vector network [30]. The idea is to construct a vector space such that words found in similar contexts in the source code are put in close proximity to one another in the vector space. The embedding table and word2vector for mapping words and tokens to values are constructed from the training code corpus consists of node types, and tokens gathered from training programs. As variable and function names and constant values can be of arbitrary lengths, we encode them as tokens (i.e., letters, symbols, and numbers).

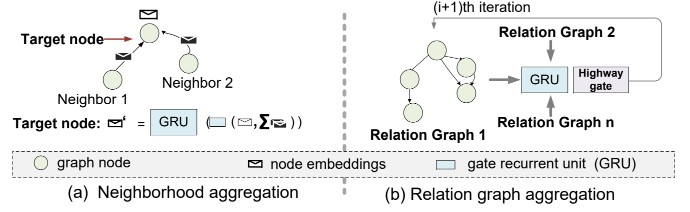


Fig. 5. Learning node embeddings by aggregating information across neighbors (a) and from other relation graphs of the same node (b). The initial node embeddings are generated using a word2vec network.

To capture the type information, we concatenate the embeddings of the (return) type of a variable, constant and function – such as `int` for an integer variable – with the AST node name representation, and pass it through a linear layer to obtain the initial representations for each node in the graph.

E. Learning over Multi-relational Graphs

Given the adjacency matrices and initial node embeddings, our multi-relational GNN generates a global one-dimensional embedding of 100 features across relation graphs.

1) Neighborhood aggregation: Like all GNNs, we use a neighborhood aggregation scheme (Figure 5a) to update node embeddings. Our 100-dimensional embedding vector, h_v, of a graph node, v, is computed by the embedding layer through recursively aggregating and transforming the representation vectors of its neighboring nodes. Nodes exchange information by sending their current state (i.e., the embedding vector) as a message to all neighbours along the edges. At each node, messages are aggregated and then used to update the associated node representation at the next embedding layer (i.e., the next iteration). After repeating this process of updating node states for a fixed number of iterations a *readout* function is used to aggregate the node states to a single embedding vector.

2) Multi-relation modeling: Unlike a standard GNN, our model propagates and aggregates information across multiple relation graphs. As depicted in Figure 5b, we achieve this by first using learnable, relation-specific functions to compute new graph states of individual relation graphs through neighborhood aggregation. We then apply a GRU cell to aggregate and update states for the same nodes across relation graphs. Formally, we use forward propagation to update the state, h_v^t, for vertex, v, of a relation graph to obtain a new state, h_v^{t+1}:

$$h_v^{t+1} := GRU(h_v^t, \sum_{\ell} \sum_{(u,v)\in A_\ell} (W_\ell * h_u^t)) \tag{1}$$

where A_ℓ are the directed edges between nodes u and v. W_ℓ and the GRU are learnable parameters. The initial node state, h_v^0, is created using word2vec as described in Sec. IV-D.

Inspired by work in natural language processing [31] that uses highway gates [32] to control the noise propagation, we also employ layer-wise highway gates to our GGNN:

$$T(h_v^{(t)}) = \sigma(h_v^{(t)} W_T^{(t)} + b_T^{(t)}) \tag{2}$$

$$h_v'^{(t+1)} = T(h_v^{(t)}) \bullet h_v^{(t+1)} + (1 - T(h_v^{(t)})) * h_v^{(t)} \tag{3}$$

where $h_v^{(t)}$ is the input to layer t+1 and obtain a new state, $h_v'^{(t+1)}$;$sigma$ is a sigmoid function; $\bullet$ is element-wise multiplication; $W_T^{(t)}$ and $b_T^{(t)}$ are the weight matrix and bias vector for gate $T(h_v^{(t)})$, respectively.

3) Readout: After we performed the neighborhood aggregation procedure across multiple embedding layers, we will obtain another set of embeddings for each token. To represent the entire program, we use a readout function to concatenate graph representations across all the neighborhood aggregation iterations and embedding layers, to form an output vector, h_G, as the global program representation of m relation graphs, G_i:

$$h_G = \text{CONCAT}\left(\sum_{i=1}^{m}\left(\left\{h_{v,i}^{(t)}|v \in G_i\right\}\right)|t = 0, 1, \ldots, n\right) \quad (4)$$

where $t = 0, 1, ..., n$, is the neighborhood aggregation iterations. Given individual node embeddings, this readout function produces the global embedding for m relation graphs.

F. Training the GNN

Our GGNN is trained *offline* using training samples from both standard vulnerability databases (CVE and NVD in this work) and open-source code examples gathered using our data collection framework described in Sec. V. The learned model can then be applied to *unseen* programs.

We train our GGNN on batched training samples, where each batch consists of positive and negative samples. As our goal is to minimize the distance between two probability distributions - predicted and actual, we choose the cross-entropy loss as our objective function. This function is proven to be a good fit for the sigmoid and softmax activation functions used by our GNN [33]. We use the minibatch stochastic gradient descent (SGD) and Adam algorithm [34] with a learning rate of 0.001. Training terminates when the loss is less than 0.005 or reaching the maximum 100 training epochs. As we will show in Sec. VII-F4, the training overhead of our vulnerability detection model is comparable to other DNN schemes. Since training is performed *offline*, it is a one-off cost and has no impact on the end-user.

G. Intuitions of FUNDED

Like most machine learning techniques, DNNs work as a black box [35], and that is just as true for our approach. Model interpretation and theoretical analysis of the working mechanism of a DNN remains an outstanding challenge and is out of the scope of this work. Xu *et al.* [36] shows that GNNs are the same powerful as the Weisfeiler-lehman test [37] in distinguishing graph structures. At a high level - as depicted in Figure 4(b) - GNNs follow a neighborhood aggregation strategy, including the *aggregate* layer and *combine* layer, which are used to iteratively update the representation of a node by aggregating representations of its neighbors. For the code example shown in Figure 1, FUNDED learns *allocation* and *deallocation* of variable `a` by aggregating its neighbor AST nodes `malloc` and `free`. The learnt information will be mapped into a numerical vector during the readout stage to allow FUNDED to learn operations performed on variable `a` across the program control and data flow graph.

V. TRAINING DATA COLLECTION

To provide large and high-quality training data for our GNN model, we leverage the available data in open-source projects by building a data collection tool.

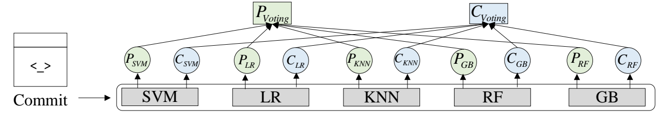


Fig. 6. Our data labeling model consists of multiple individual classifiers. For a prediction, P, given by a composition classifier, we quantify its confidence, C, which is used to filter out predictions of low confidence.

At the heart of our data collection tool is a set of expert models for predicting if a code revision is vulnerability-relevant. In essence, we form an "*expert committee*" comprised several representative classifiers (Sec. V-A). Each expert model takes as input a set of features (Table II) obtained from the commit message and code changes between commits (refer to Table I). It then predicts if the target code revision provides a patch for a vulnerability or not. All expert models are trained offline using labeled training samples. The trained models can then be applied to any *new, unseen* code commits from the *unseen* projects (Sec. V-D). In Sec. VII-E1, we compare our "mixture-of-experts" approach against alternative modeling techniques that use a single monolithic model.

A. Mixture-of-expert Model

Figure 6 depicts our model mixture, which consists of five classifiers: support vector machine (SVM), random forests (RF), k-nearest neighbor (KNN), logistic regression (LR) and gradient boosting (GB). We use these models because they have been shown to be useful in prior work [20, 23, 19, 24], but other models can be added into our expert committee too.

Unlike prior work [18] [19] [20] that directly makes use of the predictions, we first apply CP to evaluate the credential of individual classifiers for the given input to filter out predictions with high uncertainty. We then use a majority voting scheme to aggregate the remaining predictions to generate an outcome.

We describe how to build and use an expert model following the three-step process of supervised learning: (1) training data generation; (2) modeling training; (3) using the model.

B. Training Data for Expert Models

1) Collecting code revision training samples: We use the same training dataset to train each expert model. The training data are constructed from two sources. The first contains commit logs and patches reported in CVE [38] and the national vulnerability database (NVD) [39]. The second contains commit logs and patches extracted from open-source projects hosted on GitHub such as the examples given in Table I.

Logs from CVE and NVD are already associated with a known vulnerability, they can be used directly. To collect data from GitHub, we consider 1,000 top-ranked projects with the primary programming language in C or Java. We choose C and Java as they are among the most popular programming languages. However, our data collection framework is generally applicable and can be applied to other programming languages too. We apply a set of regular expression (RE) rules extended from [23] to choose commits that are likely to be vulnerability relevant.

To simplify the process for extracting vulnerability code samples, our current implementation only considers code

TABLE II
FEATURES FOR LABELING VULNERABILITY-RELATED COMMITS.

Category	Features
Project quality and activities	(1) #stars; (2) #commits; (3) # releases; (4) #contributors; (5) contribution rate; (6) #branches
Code commit description	commit message;
Code patch	code changes;

revisions that modify one source file at a time. After collecting the initial code revision samples, we manually inspect the collected data to identify if the vulnerability reported in the code commits been published in the CVE or not. If an identified vulnerability has previously been reported in CVE, we use the CVE number to establish a link with a public CVE description. Otherwise, we manually extract the code segment that contains the vulnerability, the commit logs, and the issue report (if any). We manually label *all* code revisions that have passed our RE rules to be vulnerability-relevant or not. We then use the labeled samples as our training data. In this work, we use over 3,000 manually labeled code commits (from projects with C or Java as the primary language) to train the expert models (see also Sec. V-C3 and Sec. VI-D). We stress that this manual inspection process only needs to be performed once for training the models, and the learned models can be used to gather many more samples. Therefore, the training overhead can be amortized.

2) Feature extraction: A key aspect of building a good machine-learning model is finding the right features to characterize the input. For this work, we use the three types of features given in Table II to capture the quality of the open-source project and the purpose of a code commit. Intuitively, the commit message describes the reason for a code revision - whether it is relevant to a vulnerability fixing or not, and the type of the vulnerability. The higher the quality an open-source project is, the more rigorous and meaningful a code commit message is likely to be; and an actively developed project is more likely to have regular patches for vulnerabilities.

The commit message and the modified code statements are mapped into an embeddings vector, using a pre-trained word2vec network [40]. The generated embeddings together with the feature values for the project quality and activities are put together to form an aggregated feature vector.

C. Expert Model Training

1) Training individual expert models: The training data are used to determine the optimal hyper-parameters of each expert model. Each of the training samples consists of a feature vector of numerical values and a label indicating if the code revision sample is for fixing a code vulnerability or not. For training, we simply supply the expert models with the training data and it carries out its internal supervised learning algorithm.

2) Confidence evaluation: We also apply CP to capture the "strangeness" (termed nonconformity measure) of class label y (i.e., vulnerability-relevant or not) for input x. To do so, we use a model-specific nonconformity function, $A(x, y, h)$, to estimate the nonconformity score for model h. We use the default method-specific nonconformity function given in PyCP [41]. Intuitively, unusual patterns on the feature space defined by an expert model will be given a larger nonconformity score than more common patterns.

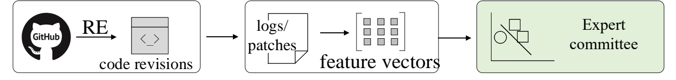


Fig. 7. Collecting and labeling open-source code samples.

To calculate the statistical confidence, we set aside 10% of the model training data as the calibration set (that is not used to train the expert models). We compute, *offline*, the calibration scores, $a_1^{y^p}, a_2^{y^p}, ..., a_n^{y^p}$, by applying function A to each of the n instances in the calibration set using the probability (y^p) given by model h for each class label, y. Given a new input, x_{n+1}, we calculate the conformity score, $a_{n+1}^{y^p}$, using function A. We then compute a *p-value*, pv, for x_{n+1} as:

$$pv = \frac{\text{COUNT}\left\{i \in \{1, ..., n+1\} : y_i = y^p \text{ and } \alpha_i^{y^p} \geq \alpha_{n+1}^{y^p}\right\}}{\text{COUNT}\ \{i \in \{1, ..., n+1\} : y_i = y^p\}} \tag{5}$$

Here, if the p-value is small (close to its lower bound $1/(n+1)$), then the prediction is very nonconforming (an outlier). If it is large (close to its upper bound 1), then the prediction is very conforming. We will only consider a prediction if its p-value is greater than $1-c$, where c is a configurable significant level (empirically set to 0.3 in this work).

3) Training overhead: The time for training the expert models is dominated by training data collection and labeling. In this work, it takes us less than three days to collect the GitHub revision samples using an automatic script (largely limited by the number of requests can be issued by Github accounts per day), and two paid annotators less than two days to manually inspect and label the collected code revisions used for training by cross-referencing the commit message, code changes and issue reports. The time in training classifiers and tuning the training data ratio is negligible (less than an hour using a multi-core server) in comparison. Since training is only performed once, it is a *one-off* cost.

D. Using the Expert Models

Once we have learned the expert models, we can use them to predict if a code commit provides a patch for a vulnerability. Figure 7 depicts the process of collecting and labeling open-source code samples. We use GitHub APIs to automatically crawl and obtain the code commits of top-ranked projects. We then apply our RE rules (see Sec. V-B1) to choose code commits that are likely to be vulnerability relevant.

Labeling code commits. For code revisions that passed our RE filters, we apply the offline trained expert models to predict if the code revision is relevant to code vulnerability fixing or not. We use the feature extractor to process the collected code commit log, patch and project-related information, to form a feature vector (as described in Sec. V-B2). Given the feature values, each expert predicts if the code revision is relevant to vulnerability fixing or not. To reach consensus among multiple classifiers (experts), we apply CP to estimate the nonconformity score of each expert's output. We keep outputs whose nonconformity scores are greater the confidence level (see Sec. V-C2). We then make the final consensus based on a simple majority voting of the remaining outputs.

TABLE III
DATASET FOR EVALUATING VULNERABILITY DETECTION.

Source	Language	#vuln. types	#samples	#positive samples
SARD & NVD	C	30	90,954	45,477
	Java	14	29,512	14,756
	Php	5	17,578	8789
GitHub	C	10	10,400	5,200
	Swift	5	2,506	1253

TABLE IV
CODE REVISION HISTORY DATASET.

Source	Language	# commits	#vulnablility related commits
GitHub	C	5,718	2,573
	Java	2,195	1,796
SAP	Java	1,787	804
ZvD	C/C++	3,422	1,540

Extracting code samples. For each code commit that passed our RE filters, we use code changes to locate the previous version of a patched function. We then extract the code of this function and associate it with the label (vulnerable or not) given by the expert committee.

Continuous learning. One of the advantages of using CP to evaluate an expert's confidence is that we can use samples with low credibility to improve an expert model over time. Doing so allows us to continuously improve expert models over time. In Sec. VII-G2, we demonstrate how continuous learning can be utilized to improve our data collection framework.

VI. EXPERIMENTAL SETUP

A. Evaluation Datasets

We evaluate FUNDED on two types of datasets. We evaluate our vulnerability detection model (Sec. IV) on code samples written in four source languages: C, Java, Php and Swift. We test the mixture-of-expert approach (Sec. V), the core of our data collection tool, on code revision history of projects using C, C++ and Java as the primary programming languages.

Dataset for vulnerability detection. Table III gives details of this dataset, which contains a total of 150,950 samples at the function level with source languages in C, Java, Php and Swift. Half of our samples are positive (vulnerable) code samples. We restrict our scope to the top-5 to top-30 most dangerous software errors defined in CWE 2019 (e.g., "buffer overflow", "out-of-bounds read/write", "NULL pointer dereference"). We construct this dataset from SARD [42], NVD [39] and open-source projects hosted on GitHub. Like prior work [5], we use the patched version provided by SARD and NVD as a negative (or vulnerable-free) code sample. Similarly, for the vulnerable samples collected from GitHub, we apply the corresponding patch commit to obtain the vulnerability-free version. Our test samples contain realistic code samples with thousands lines of code. More information and examples can be found from the supplementary documents. The supplemental document gives the distribution for each CWE type used in the evaluation and an example of our test cases.

Code revision history. As can be seen from Table IV, this dataset includes a total of 6,713 vulnerability-relevant code revisions from GitHub, and the SAP [43] and ZvD [44] datasets. Among the 4,369 vulnerability-relevant commits from GitHub, 2,071 are established via CVE and NVD links; the remaining 2,298 are obtained from the top-1000 most popular projects on GitHub with C and Java as the primary programming languages. For the latter, we manually examined and labeled the commits to establish the ground-truth. Note that the SAP and ZvD datasets already contain negative samples (i.e, randomly chosen vulnerability-irrelevant commits collected from the same project where a vulnerability-relevant code commit is found). We apply the same methodology to obtain negative commits from GitHub. Specifically, we retain code commits that have passed our RE filters but are found out to be vulnerability-irrelevant through manual inspection. Overall, we have a total of 13,122 code commit samples, containing both vulnerability-relevant and irrelevant commits.

B. Competitive Approaches

For vulnerability detection, we compare FUNDED to six relevant methods: VULDEEPECKER [5], μVULDEEPECKER [6], LIN *et al.* [3], VUDDY [4], DEEPBUGS [16], and DEVIGN [15]; the first three build upon BiLSTM, VUDDY uses hash functions to discover vulnerable code clone, DEEPBUGS utilizes a feedforward neural network for bug detection, and DEVIGN uses a standard GNN [45] operating on graph representation with untyped AST edges. All but μVULDEEPECKER of the competitive schemes make a binary decision to predict if the code contains a bug or vulnerability or not.

For data collection, we compare FUNDED with five SOTA data collection methods: VCCFINDER [18], SABETTA *et al.* [20],VULPECKER [19], ZVD [24] and ZHOU *et al.* [23].

C. Implementation

We implement GNN-based vulnerability detection model using Tensorflow v.1.8 [46] and models for data collection using the Python scikit-learn package [47]. To construct the AST, we use Soot [48] for Java, ANTLR [49] for Swift, Php and Joern [50] for C/C++. We train and test all approaches on a multi-core server with a 14-core 2.4 GHz Intel Xeon CPU and an NVIDIA 2080Ti GPU.

D. Evaluation Methodology

Model evaluation. Unless stated otherwise, we use *five-fold cross-validation* to evaluate all approaches on their respective dataset. This standard methodology evaluates the generalization ability of a predictive model.

Performance report. We use four *higher-is-better* metrics:

***Accuracy*:** The ratio of correctly labeled cases to the total number of test cases.

***Precision*:** The ratio of correctly predicted samples to the total number of samples that are predicted to have a specific label. This metric answers questions like "*Of all the code revisions that are labeled to be vulnerability-relevant, how many are actually correct?*". High Precision indicates a low *false-positive* rate.

***Recall*:** The ratio of correctly predicted samples to the total number of test samples that belong to a class. This metric answers questions like "*Of all the vulnerable test samples, how many are actually labeled to be vulnerable?*". High recall suggests a low *false-negative* rate.

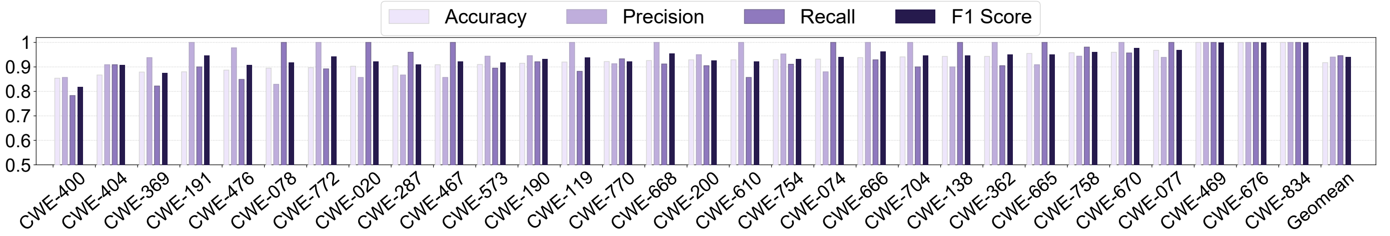


Fig. 8. FUNDED delivers on average, an accuracy of 92%, for detecting C functions with the top-30 CWE vulnerabilities.

***F1 score*:** The harmonic mean of Precision and Recall, calculated as $2 \times \frac{Recall \times Precision}{Recall + Precision}$. It is useful when the test data have uneven distribution of vulnerability types.

We report the *geometric mean* of the aforementioned evaluation metrics across the cross-validation folds because it is widely seen as a more reliable performance metric over the arithmetic mean [51].

VII. EXPERIMENTAL RESULTS

Highlights of our evaluation results are:

- FUNDED delivers, on average, a 92% accuracy, for software vulnerability detection (Sec. VII-A and Sec. VII-B).
- FUNDED outperforms all competing methods for detecting (Sec. VII-C) and collecting (Sec. VII-E vulnerable code).
- We provide detailed analysis for the working mechanisms of FUNDED (Sec. VII-G and Sec. VII-F).

A. Overall Results

In this experiment, we apply FUNDED to detect vulnerabilities of C functions with the top-30 CWE vulnerability types, and we evaluate on other languages in Sec. VII-D. In this experiment, we train our detection model using both training samples from standard vulnerability databases and those collected by our data collection tool. Figure 8 reports evaluation metrics for each vulnerability type. FUNDED successfully identifies most of the vulnerable samples with an average Accuracy and Precision of 92%. High precision reduces the false-positive rate and is important in practice because false-positive results waste developers' time for verification. FUNDED also has a high Recall and F1 score of 0.94 (up to 0.99), indicating that it has a low false-negative rate and rarely misses vulnerabilities. FUNDED gives less than 90% Accuracy (but still above 80%) on some CWE types like CWE-400 and CWE-369 due to the word2vec model (used for initial node embeddings - see Sec. IV-D) is less accurately in capturing tokens for API misuse. In future, we could enhance FUNDED with a more powerful language model. Finally, we note that FUNDED can identify vulnerabilities from real-world, sophisticated code where other competing methods fail.

B. Evaluation on Large Code Bases

We apply FUNDED to five open-source projects that were also evaluated in prior studies [5, 18, 52, 53]. Table V lists the software versions and the number of function-level vulnerabilities. We exclude code from these projects in training to ensure the trained model is tested on "*unseen*" programs.

Figure 9 summarizes the successfully identified individual vulnerabilities for each project, where bars on the left-hand side are the total number of successfully discovered vulnerabilities. Here, a dark symbol means a vulnerability is identified by a model, where a circle means the vulnerability is reported in NVD or CVE while a square indicates the vulnerability is not reported in the two databases – also known as "silently-patched vulnerabilities" [5].

TABLE V
EVALUATION DATASET OF FIVE OPEN-SOURCE PROJECTS.

No.	Project	Versions	# vuln.
1	FFmpeg	v3.4.1, v3.4.2, v4.0.1, v4.1	12
2	ImageMagick	v7.0.6, v7.0.8	14
3	Linux kernel	v4.19 , v4.20 , v5.0, v5.4	16
4	OpenSC	v1.8.2, v0.19.0	8
5	rdesktop	v1.8.2	6

FUNDED outperforms all competing approaches by identifying 53 out of 56 vulnerabilities, including 11 "silently-patched" vulnerabilities, with a Recall of 0.95. By operating on a graph representation, DEVIGN outperforms the sequence models with higher overall accuracy, showing the advantage of a GNN-based model. However, DEVIGN also fails to detect 12 vulnerabilities that a sequence model succeeds because it sacrifices many of the syntactic and semantic relationships when trading sequential for an untyped graph representation. FUNDED improves over DEVIGN by identifying 13 more vulnerabilities, giving 33% improvement in Accuracy and Recall. Moreover, FUNDED identifies four vulnerabilities that all other models fail to detect. There is one each case in FFmpeg, ImageMagick and Linux kernel that FUNDED fails to detect but can be identified by others. Those are cases where the vulnerability is caused by the misuse of API parameters. Such patterns are not captured by word2vec model used by FUNDED. This issue can be tackled by using a better language embedding model, for which we leave as future work.

C. Vulnerability Detection on Individual Datasets

We now evaluate our vulnerability detection model on the individual datasets in Table III using cross-validation.

1) Evaluation on standard datasets : This experiment applies all approaches to C functions from SARD and NVD. In Sec. VII-D, we extend the evaluation to Java, Php and Swift.

Figures 16 and 10 show that FUNDED delivers the best overall performance for vulnerability detection. VUDDY and DEEPBUGS give low detection accuracy due to the limitations of their detection models. Using BiLSTM, VULDEEPECKER and μVULDEEPECKER are effective in a small number of vulnerability types but gives an accuracy of less than 50% for certain types (like CWE-469, CWE-676, and CWE-834) and can lead to a large number of false positives (i.e., low precision). By leveraging a rich set of *manually labeled*

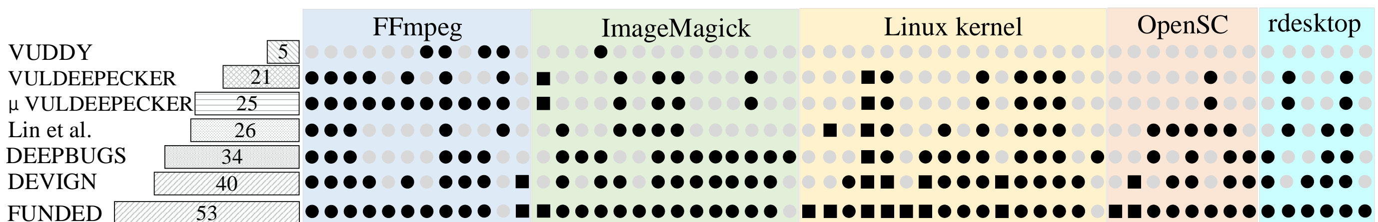


Fig. 9. The number of vulnerabilities identified by each approach for each open-source project. A solid symbol represents a successfully detected vulnerability, where a circle means the vulnerability is reported in NVD or CVE, while a square means the vulnerability is not reported in the standard databases. FUNDED successfully detects more vulnerabilities than others.

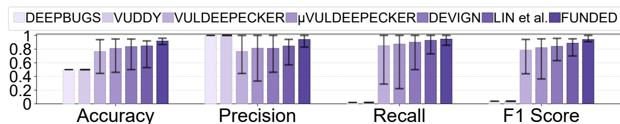


Fig. 10. Evaluation on standard vulnerability databases. Min-max bars show performance across vulnerability types.

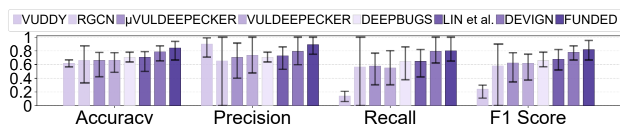
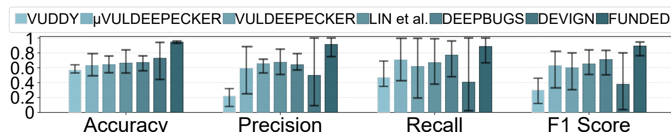


Fig. 11. Evaluation on GitHub samples. FUNDED gives the best Accuracy, Recall and F1 score.

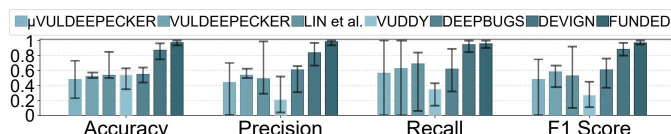
training data, LIN *et al.* is the best-performing competing method. However, its BiLSTM-based model fails to detect some vulnerable cases. For test cases with a CWE-834 vulnerability, LIN *et al.* only successfully discovers 52.6% of the vulnerable samples. DEVIGN gives the second-highest overall accuracy, but that only translates to a marginal improvement of less than 2% over μVULDEEPECKER and is outperformed by LIN *et al.* DEVIGN also gives low performance several vulnerability types like CWE-138 and CWE-754, while other models have an accuracy of over 80%.

For the majority of the vulnerability types, FUNDED outperforms all other methods across all evaluation metrics. In a handful of types of vulnerabilities, FUNDED misses one vulnerable sample that can be detected by the *best-performing alternative* method. Most of such cases are because our the pre-trained word2vec network does not capture a certain language keyword, e.g., `sizeof`. This can be improved by explicitly adding important keywords to the network vocabulary so that word2vec can directly model the semantics of those keywords at the word (instead of token) level [54]. Overall, FUNDED is the only scheme that delivers averaged accuracy of over 90% and has the best overall performance across evaluation metrics.

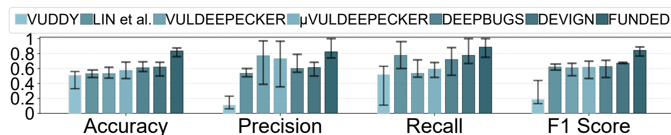
2) Evaluation on GitHub dataset: We now extend our experiments to C functions collected from GitHub. In this experiment, we train on the SARD-NVD datasets and test on the GitHub dataset. Figure 11 reports results across metrics, where the min-max bar shows the variance across vulnerability types. As expected, training data from the standard databases cannot fully represent the vulnerable code samples seen in real-life programs. As a result, we see a drop in accuracy for GitHub code samples. Overall, FUNDED delivers the best performance for Accuracy, Recall, and F1 score. While VUDDY has the best Precision, it has a much lower Recall. This is because although VUDDY has the lowest false-positive rate, it is too restricted in detecting vulnerabilities – it misses over 85% of the vulnerable testing samples. LIN *et al.* is the best-performing alternative model for Accuracy, but its F1 score is 20% lower than that of FUNDED, suggesting that FUNDED achieves a better balance for false and negative positives.



(a) C to Java



(b) Java to Php



(c) C to Swift

Fig. 12. Apply transfer learning to port a detection model for a new programming language.

D. Cross-languages Learning

Prior work in other domains has shown that neural networks trained on similar inputs for different tasks often share useful commonalities [55]. The observation is that the properties of the input that are abstracted by the beginning layers of the neural networks are mostly independent of the task. By contrast, information learnt at the last few layers in networks is more specialized towards a specific task. Our work exploits this observation to reuse these parts of the network learned from one language to speed up the learning for a new language. Furthermore, by modeling the program graph structure rather than the surface level information at the code level, our approach can better capture the vulnerable code structure across programming languages too. This suggests that we can port a detection model for a new programming language by leveraging the knowledge of the vulnerable code patterns learned from another language. The technology for achieving this is called *transfer learning* [56, 57]. Since we use the same network structure, transfer learning is achieved by copying the weights of a model built for one language to initialize the network for another. Then, we train the model as usual using a small set of training samples for the target language.

```
1  ...
2  if (inputStream != null) {
3      inputStream.close();
4  }
5  if (fileOutputStream != null) {
6      return false;
7  }
8  fileOutputStream.close();
9  return false;
10 ...
```

(a) Java code sample from GitHub with CWE-775

```
1  ...
2  if ($userfile != FALSE){
3    while (($user = fgetcsv($userfile)) !=
         FALSE)
4     if ($user[0] == $username)
5       return TRUE;
6    return FALSE;
7  }
8
9  fclose($userfile);
10 ...
```

(b) Php code sample from Github with CWE-775

Fig. 13. Both the C code (a) and Php code (b) contain an “improper resource shutdown” vulnerability. Although the programs were written in two different languages and are from two different open-source repositories, they have similar control and data flows that lead to a common vulnerability.

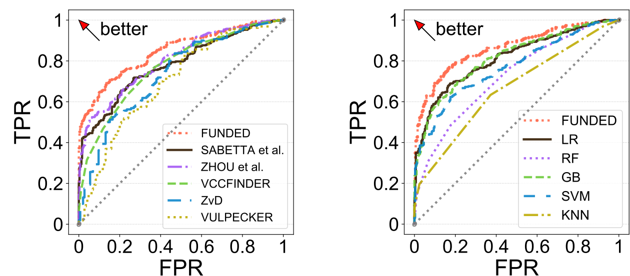
To illustrate how graph-level knowledge can be reused across programming languages for code vulnerability detection, consider the two code segments in Table 13 from two different repositories hosted on GitHub. Both code samples contain an *improper resource shutdown* vulnerability where the program does not release a file handler after its effective life-cycle ends. Specifically, the Java function in Figure 13 does not close the file handler if `fileOutputStream` is not `null`, and the Php method does not close the file handler if `userfile` is `True`. By modeling the subtle program structures that lead to a vulnerability, our approach can reuse the knowledge learnt from the Java training samples to model the same vulnerability for Php programs or vice versa.

In this experiment, we first train a baseline model for one language. We then apply transfer learning to port the baseline model to another language using cross-validation. We consider three cross-language settings: C to Java, Java to Php and C to Swift, where the first is the language the baseline model is trained for, and the second is the new language to be targeted. Figure 12 shows that FUNDED can better utilize prior knowledge to detect software vulnerabilities for a new programming language, by delivering the best performance across language settings and evaluation metrics. This is because FUNDED captures much of the language-agnostic information for vulnerable code patterns. This feature is useful for languages or libraries with scarce training samples.

E. Evaluation of Data Collection

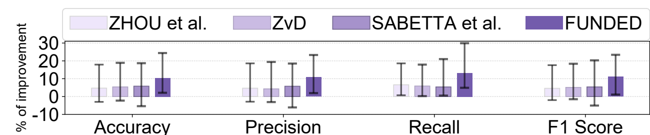
We now evaluate our data collection framework on the code revision datasets given in Table IV using cross-validation.

1) Compare to alternative data labeling approaches: Figure 14 shows the ROC curves for different data collection methods and individual expert models (Sec. V-A) used in this work. The ROC diagram plots the true-positive rate (TPR) against the false-positive rate (FPR) at different classification thresholds, where a positive sample refers to a vulnerability-relevant commit. Lowering the classification threshold (i.e., a higher FPR) increases the likelihood for labeling more samples as vulnerability-relevant, thus increasing both true and false positives. FUNDED delivers the best overall performance under a meaningful FPR threshold (e.g, less then 0.5), by giving a curve closer to the top-left corner than other methods.

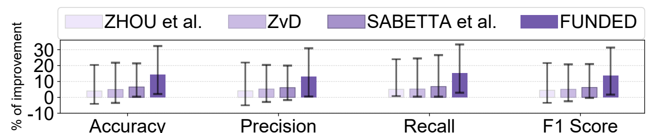


(a) Comp. to data collect. methods. (b) Comp. to individual experts

Fig. 14. ROC curves for true positive rate (TPR) and false positive rate (FPR). We compare FUNDED to other automatic data collection frameworks (a) and individual expert models used in our expert committee (b). Methods that give curves closer to the top-left corner indicate a better performance.



(a) Improvement over the baseline VULDEEPECKER model.



(b) Improvement over the baseline μVULDEEPECKER model.

Fig. 15. Performance improvement over the baseline code detection models when using extra labeled training samples from GitHub. The min-max bar shows the range of improvement across the top-10 CWE vulnerability types.

2) Impact of collected training data: In this experiment, we evaluate if the open-source project samples help in learning a better detection model. To isolate the impact of our GNN model, we test if the training samples gathered by a data collection method can improve the baseline model of VULDEEPECKER [5] and μVULDEEPECKER [6]. We use the SARD-based training datasets [42] to learn the baseline model. Next, we include additional 400 GitHub code samples (with an equal positive-negative split) that are collected by FUNDED and other methods [23, 20, 24] to the training dataset to learn a second, refined model. We then apply both the baseline and the refined models to 1,000 C test samples from Github, where half of the samples are vulnerable code. For a fair comparison, we ensure that the test data only contain the same CWE types seen in the training data of the baseline model.

Figure 15 reports the performance improvement by using a data collection approach. The min-max bar shows the range of improvement across different vulnerability types, where a negative value suggests a decrease in performance. Using extra training data from GitHub could improve the baseline models. FUNDED delivers the best and consistent improvement across all evaluation metrics, outperforming the competitive

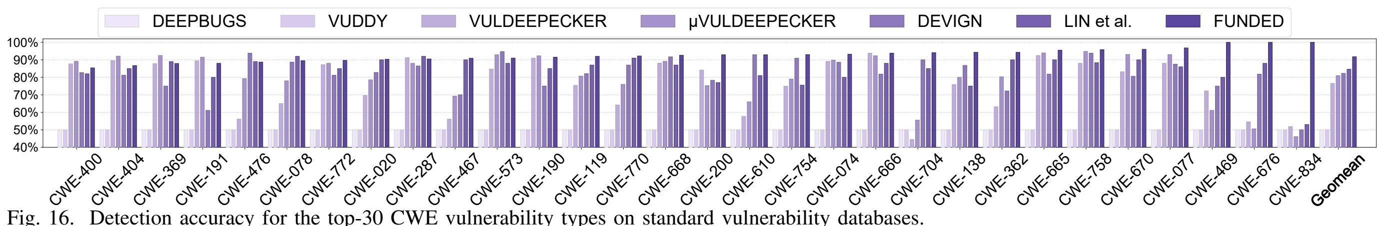


Fig. 16. Detection accuracy for the top-30 CWE vulnerability types on standard vulnerability databases.

TABLE VI
TOP-3 ACCURACY FOR PREDICTING THE VULNERABILITY TYPE.

Method	Accuracy	Method	Accuracy
μVULDEEPECKER	78.1%	DEVIGN	81.4%
RGCN	79.0%	**FUNDED**	**93.8%**

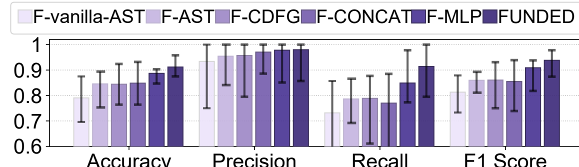


Fig. 17. Comparing implementation variants of FUNDED. Our implementation gives the best overall performance.

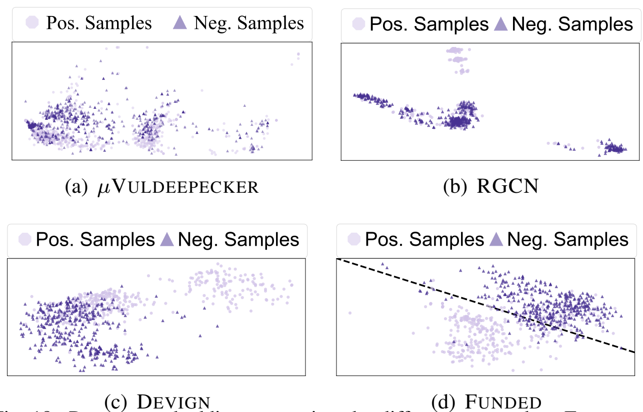


Fig. 18. Program embedding space given by different approaches. FUNDED is more effective in mapping code samples into a space where a more distinctive boundary can be drawn to separate vulnerable and benign code samples.

methods with at least 4.6% (up to 10.2%) improvement with higher quality training data. For the 200 GitHub samples to be labeled as vulnerability-relevant, the mixture-of-expert model of FUNDED has an accuracy over 90%, while others have a lower accuracy between 60.1% and 81.6%. The incorrectly labeled examples given by some competing models could also have a negative impact on the resulting performance. This experiment shows the need of having an accurate data labeling model to collect additional, high-quality training examples. FUNDED offers exactly such capabilities.

F. Analysis of Vulnerability Detection Model

1) Predict the vulnerability type: So far, we have applied FUNDED to make a binary decision to predict if a piece of code contains a vulnerability or not. In this experiment, we extend FUNDED to predict the types of vulnerabilities. We compare FUNDED to μVULDEEPECKER, the only multi-class vulnerability detection model in the competitive schemes, but we also extend two GNN-based variants: RGCN [58] and DEVIGN to multi-class predictions. We use the C datasets from SARD, NVD and GitHub (see Table III) for evaluation.

Table VI reports the top-3 accuracy of four methods. This metric checks if one of the top-3 predicted labels (ranked based on prediction probabilities) matches the ground-truth of a testing sample. This in practice means the developer only needs to verify three potential vulnerabilities. As expected, we see a drop in accuracy from binary prediction to the multi-class prediction. However, FUNDED remains the best-performing model and is the only one that gives an accuracy of over 90%.

2) Impact of implementation choices: We compare FUNDED to several implementation variants using the SARD dataset (that isolates the impact of our data collection method). The first variant, referred to as F-vanilla-AST, operates on the standard AST (without the additional edges describes in Sec. IV-C). The second variant, referred to as F-AST, operates on the augmented AST but does not have the control and data flow edges. The third variant, referred to as F-CDFG, operates on an augmented AST but with only additional control and data flow edges. The fourth variant, referred to as F-CONCAT, learns individual embeddings for each relation graph and then concatenates them for prediction. The last variant, referred to as F-MLP, uses a multilayer perceptron (MLP) layer to learn and aggregate embedding of individual relation graphs [26], but without attention and highway gates.

Figure 17 reports that using the standard AST is inadequate for modeling the program structures. By augmenting the AST with control and data flow information, F-AST or F-CDFG modestly improves the accuracy of F-vanilla-AST by 5%. However, using the AST or CDFG alone is insufficient, as both give an accuracy of less than 85%. F-CONCAT also gives lower performance compared to FUNDED, suggesting that simply combining the embeddings of relation graphs is less effective. This experiment reinforces the importance of utilizing and aggregating the information of the additional control and data flow information. By operating on multiple relation graphs, F-MLP is the best-performing competitive approach, showing the great advantage of multi-relational learning. FUNDED further improves F-MLP by employing an attention mechanism and using highway gates to minimize the diminishing gradient issue when modeling long dependence.

3) Program embedding space: To illustrate the learned program representation, we visualize the embedding space of testing data. Intuitively, an effective classifier should map the testing inputs into space where distinct decision boundaries between the positive and negative classes can be drawn.

Figure 18 shows how a trained BiLSTM (that is used by VULDEEPECKER, μVULDEEPECKER, and LIN *et al.*), DEVIGN (a standard GNN), RGCN and FUNDED map more than 1,000 test samples of C functions from NVD, SARD, and GitHub (with an equal positive-negative split) onto the embedding space. To aid clarity, we apply t-SNE [59], a visualization

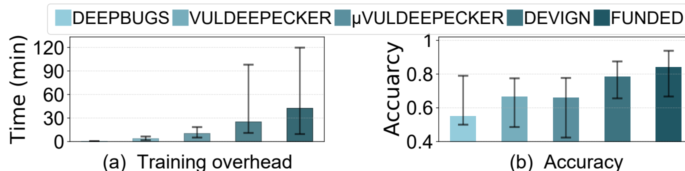


Fig. 19. Training overhead (a) and accuracy (b). The min-max bars show the variances across evaluation settings.

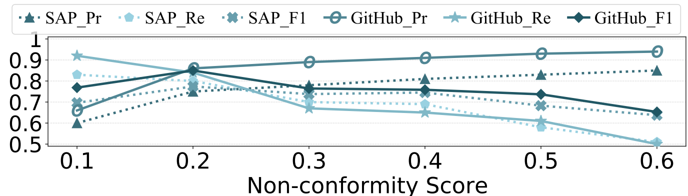


Fig. 20. How performance of our data labeling model changes as the non-conformity threshold increases on the SAP and Github datasets.

technique, to project the multi-dimensional embedding space into a two-dimensional space. Compared with other methods, FUNDED (Figure 18b) is more effective in mapping the test samples into space where most of the samples can be separated by a binary decision boundary. The result shows FUNDED is more effective in extracting essential program structures for vulnerability detection.

4) Model training overhead: Figure 19 shows the training overhead and the achieved accuracy for different vulnerability detection models. The training time was measured by applying all approach to the largest training dataset used in this work, which consists of 101,354 C functions from SARD, NVD, and Github as described in Table III. Training terminates when the loss does not improve within 20 consecutive training epochs, or reaches a 95% accuracy on the valuation set, or meets the termination criteria given in the published implementation. This experiment was conducted on a multi-core server using a desktop level NVIDIA 2080Ti GPU.

All the models can converge within two hours on our training datasets. DEEPBUGS incurs the least training time because it uses a simple feedforward neural network. However, DEEPBUGS delivers the lowest accuracy because the model is inadequate in capturing complex code structures. The remaining DNN methods require longer training time than DEEPBUGS, but they give significantly better detection accuracy. FUNDED incurs longer training time compared to other DNN methods because the model learns across multiple relational graphs. However, the training overhead of FUNDED is still comparable to other DNN models (within two hours), but it yields better detection accuracy. We stress that the model training is performed offline and is a *one-off cost*. Furthermore, our approach can make a prediction within seconds, which thus has a negligible impact on the end-user.

G. Analysis of Data Collection Framework

1) Confidence evaluation: This experiment evaluates how often our CP function (see Sec. V-C) successfully detects when a data labeling prediction is wrong. We apply our approach to the code revision history dataset (Table IV). Our CP scheme successfully catches 91% of the inputs when an expert model gives a wrong prediction, and has a low false positive (i.e., when the CP model thinks the classifier is wrong but it is not) rate of 9%. Note that we can also apply CP to improve our detection model further. We found that an unoptimized CP is able to catch 80% of the cases when our detection model gives a wrong prediction for vulnerable code.

2) Continuous learning: In this experiment, we check if one can use the ground-truth labels for predictions flagged by the CP to update a classifier. To do so, for predictions with a p-value greater than 0.7 (see Sec. V-C2), we manually check the code revision to obtain the ground-truth. By using the ground-truth for only 5% of the mispredicted samples to update classifiers, our data labeling model can achieve over 90% for F1 score on the testing data. This translates into an improvement of over 35% for the initial labeling model. Thus, using CP as a confidence measurement provides a practical way to gradually improve our data collection framework.

3) Impact of non-conformity threshold: Figure 20 shows how the nonconformity threshold affects the performance of our data labeling model on code commits collected from the GitHub and the SAP datasets. A larger nonconformity threshold can help in reducing the false-positive rate, leading to higher Precision, but it can also negatively affect the Recall and F1 score. In this work, we use 0.3 for the nonconformity threshold as it gives the best overall performance.

VIII. DISSCUSSION AND FUTURE WORK

FUNDED is among the first attempts in employing GNNs and CP for code vulnerability detection. Naturally, there is room for future work and further improvement. We discuss a few points here

Model robustness. Machine learning models can suffer from adversarial attacks where carefully constructed data samples can lead to bias and deviant behavior of a trained model. However, we believe that FUNDED is less susceptible to this problem for several reasons. First, we use code samples collected from large-scale open-source repositories to improve the coverage and diversity of our training dataset. Injecting adversarial samples into thousands of top-ranked, actively-maintained opensource projects without being noticed by the developers and users is highly unlikely, and even if this is achievable, it will incur significant efforts. Therefore, the complexity of the attack is high. Secondly, since our data collection framework uses multiple models, launching an attack to N models would typically increase the complexity of the attach by N times because the adversarial examples are often tightly coupled with the target model [60]. Finally, our CP model can be useful in filtering out adversarial samples by employing a threshold-based analysis which was shown to be effective in defending against adversarial samples [61, 62].

Model capability. The capability of GNNs is limited by the depth and width of the model [63]. Our work targets at the function level vulnerability detection, and we found that our GGNN is sufficient in learning program structures. However, we envision that to model larger programs will need a larger and deeper GGNN. To train a larger neural network typically also requires a larger amount of training data for which our data collection framework will be useful. It would also be

interesting in applying neural architecture search techniques [64] to find the right neural network structures. We leave this as our future work.

Language model. In this work, we use word2vec to initialize the node embeddings by learning from the surface-level syntax information (Sec. IV-D). The pre-trained word2vec model was originally designed for natural language processing and is not tuned for exploiting the well-defined structures of a programming language. As a result, it could miss some important language keywords. In this work, we have tried to circumvent this issue by explicitly adding some important keywords to the network's vocabulary. This workaround requires manual intervention and hence does not scale well to new programming languages. A better language model designed for modeling program language structures can alleviate this issue by eliminating expert involvement. Our future work will explore a language model that is specifically built for modeling program source code like code2vec [65].

Model interpretability. Machine learning techniques, in general, have the problem of relying on black boxes. Theoretical analysis for the capability and boundaries of a DNN is currently an active research field [36, 63, 66]. Providing a theoretical proof of the underlying working mechanism of FUNDED is our future work. One way to gain insight into why the model fails to produce the desired result is to train an interpretable model (or the so-called surrogate models) like linear regressor to approximate the predictions of the underlying black-box model [65].

IX. Related Work

Our work builds upon the past foundations of code vulnerability analysis, machine learning and software engineering. We leverage the recent development in graph neural networks to model the program graph structures to learn vulnerable code patterns and use the learned knowledge to detect code vulnerabilities. Our work also exploits ensemble learning to collect training samples from real-life open-source projects to provide additional data for training the code vulnerability detection model.

Classical approaches. Early work in software vulnerability detection relies on expert-crafted rules [67] . However, it is not trivial to construct high-quality rules as this requires heavily expert involvement. Symbolic execution [68, 69] sidesteps the need for hand-crafted rules by exploiting symbolic values and analyzing their use over the control flow graph of a program on source code, but it does not scale to large programs and often suffer from high false positives [70].

Deep learning based vulnerability detection. Our work is part of the recent efforts in DL-based software vulnerability detection [5, 6, 3, 71]. A comprehensive review of the field can be found at [7]. Prior work in the area treated source code or the AST as a sequential string and often ignores the structural information of the program. Our work exploits and extends GGNN to better model programs with multiple relationships. To provide sufficient training data, some of the recent work develop predictive models to automatically extract vulnerable code samples from code revision history [43, 20, 24]. Inspired by these recent efforts, FUNDED combines probabilistic learning and statistical assessment to automatically extract vulnerable code samples from open-source projects, significantly improving the quality of the extracted samples.

Graph neural networks. GNNs have shown promising results in processing graph data structures for tasks like mining social networks [12], entity alignment [13], and binary similarity detection [14]. DEVIGN [15] is most closely related to our work. Different from DEVIGN, our approach simultaneously models multiple code relationships accuracy and is more effective for cross-languages learning. The GNN presented in this paper extends our recent work on using multi-relational graphs to model program structures [26]. The new GNN advances [26] by extending the AST encoding to capture additional code relationships and type information and using GRU and highway gates to model longer-term dependencies, leading to better performance as shown in Sec. VII-F2.

Machine learning based software development. There is an increasing interest in applying machine learning techniques to software development [72]. Existing approaches address a variety of development tasks, including fuzz testing [73, 74], detecting code clone [75, 4, 19, 76], improving static analysis for bug funding [77, 78], repairing programs [79], defect prediction [80, 81], attack detection [82] and processing bug reports [18, 23, 24]. FUNDED builds on those past foundations but is quality different from these studies.

X. Conclusion

We have presented FUNDED, a novel graph-learning-based approach for learning code vulnerability detection models. FUNDED extends the standard graph neural network to model multiple code relationships that are essential for modeling program structures for vulnerability detection. To provide sufficient training data, FUNDED combines probabilistic learning and statistical assessments to *automatically* collect vulnerable code samples from open-source repositories. We apply FUNDED to detect source code vulnerabilities at the function level on large real-life datasets. Experimental results show that FUNDED significantly outperforms a wide range of competitive approaches across evaluation metrics.

References

[1] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *Proceedings of the IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1744–1772, 2018.

[2] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proceedings of the 2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2017, pp. 1298–1302.

[3] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, 2019.

[4] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[5] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *Proceedings of the NDSS*, 2018.

[6] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, 2019.

[7] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.

[8] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *Proceedings of the arXiv preprint arXiv:1807.06756*, 2018.

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

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1981, pp. 207–218.

[11] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *Proceedings of the IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[12] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 1416–1424.

[13] Y. Wu, X. Liu, Y. Feng, Z. Wang, and D. Zhao, "Jointly learning entity and relation representations for entity alignment," *Proceedings of the arXiv preprint arXiv:1909.09317*, 2019.

[14] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[16] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[17] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[18] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[19] 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 *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 201–213.

[20] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 579–582.

[21] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Proceedings of the Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[22] G. Shafer and V. Vovk, "A tutorial on conformal prediction," *Proceedings of the Journal of Machine Learning Research*, vol. 9, no. Mar, pp. 371–421, 2008.

[23] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 914–919.

[24] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss," in *Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 485–492.

[25] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *Proceedings of the arXiv preprint arXiv:1508.01991*, 2015.

[26] G. Ye, Z. Tang, H. Wang, D. Fang, J. Fang, S. Huang, and Z. Wang, "Deep program structure modeling through multi-relational graph-based learning," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 111–123.

[27] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *Proceedings of the ICLR*, 2015.

[28] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[29] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proceedings of the ICLR*, 2018.

[30] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the Advances in neural information processing systems*, 2013, pp. 3111–3119.

[31] A. Rahimi, T. Cohn, and T. Baldwin, "Semi-supervised user geolocation via graph convolutional networks," in *Proceedings of the ACL*, 2018".

[32] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," *Proceedings of the arXiv preprint arXiv:1505.00387*, 2015.

[33] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Proceedings of the Advances in neural information processing systems*, 2018, pp. 8778–8788.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Proceedings of the arXiv preprint arXiv:1412.6980*, 2014.

[35] F. Pasquale, *The black box society*. Harvard University Press, 2015.

[36] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *Proceedings of the ICLR*, 2018.

[37] B. Weisfeiler and A. A. Lehman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Proceedings of the Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.

[38] "Common Vulnerabilities and Exposures (CVE)," https://cve.mitre.org/.

[39] "National Vulnerability Database (NVD)," https://nvd.nist.gov.

[40] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *Proceedings of the arXiv preprint arXiv:1402.3722*, 2014.

[41] V. N. Balasubramanian, A. Baker, M. Yanez, S. Chakraborty, and S. Panchanathan, "Pycp: an open-source conformal predictions toolkit," in *Proceedings of the IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, 2013, pp. 361–370.

[42] NIST, "Software Assurance Reference Dataset Project," https://samate.nist.gov/SRD/.

[43] "SAP Dataset," https://github.com/SAP/vulnerability-assessment-kb/tree/master/MSR2019.

[44] "Zvd Dataset," https://github.com/SecretPatch/Dataset.

[45] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, in *Gated graph sequence neural networks*, 2016.

[46] "TensorFlow." https://www.tensorflow.org/.

[47] "Scikit-learn: Tools for predictive data analysis," https://scikit-learn.org.

[48] "Soot: A framework for analyzing and transforming Java applications," http://sable.github.io/soot/.

[49] "ANTLR (ANother Tool for Language Recognition) ," https://www.antlr.org/.

[50] "Joern(Open-Source Code Querying Engine for C/C++.)," https://joern.io/.

[51] W. Ertel, "On the definition of speedup," in *Proceedings of the International Conference on Parallel Architectures and Languages Europe*. Springer, 1994, pp. 289–300.

[52] A. Younis, Y. Malaiya, C. Anderson, and I. Ray, "To fear or not to fear that is the question: Code characteristics of a vulnerable functionwith an existing exploit," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 97–104.

[53] S. Ognawala, R. N. Amato, A. Pretschner, and P. Kulkarni, "Automatically assessing vulnerabilities discovered by compositional analysis," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, 2018, pp. 16–25.

[54] T. Mikolov, S. Kombrink, L. Burget, J. Černockỳ, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Proceedings of the 2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2011, pp. 5528–5531.

[55] M. Long, H. Zhu, J. Wang, and M. I. Jordan, "Deep transfer learning with joint adaptation networks," in *Proceedings of the International conference on machine learning*. PMLR, 2017, pp. 2208–2217.

[56] S. J. Pan and Q. Yang, "A survey on transfer learning," *Proceedings of the IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[57] L. Torrey and J. Shavlik, "Transfer learning," in *Proceedings of the Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 2010, pp. 242–264.

[58] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.

[59] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Proceedings of the Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[60] D. Hendrycks and K. Gimpel, "Early methods for detecting adversarial images," *Proceedings of the arXiv preprint arXiv:1608.00530*, 2016.

[61] S. Kokalj-Filipovic, R. Miller, and G. Vanhoy, "Adversarial examples in rf deep learning: Detection and physical robustness," in *Proceedings of the 2019 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2019, pp. 1–5.

[62] D. Meng and H. Chen, "Magnet: a two-pronged defense against adversarial examples," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 135–147.

[63] A. Loukas, "What graph neural networks cannot learn: depth vs width," *Proceedings of the arXiv preprint arXiv:1907.03199*, 2019.

[64] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[65] M. T. Ribeiro, S. Singh, and C. Guestrin, """why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[66] R. Sato, "A survey on the expressive power of graph neural networks," *Proceedings of the arXiv preprint arXiv:2003.04078*, 2020.

[67] "Findbugs," http://findbugs.sourceforge.net/.

[68] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Proceedings of the Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[69] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the 24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 49–64.

[70] K. Wang and Z. Su, "Learning blended, precise semantic program embeddings," in *Proceedings of the PLDI*, 2020.

[71] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[72] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[73] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 50–59.

[74] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[75] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[76] T. Unruh, B. Shastry, M. Skoruppa, F. Maggi, K. Rieck, J.-P. Seifert, and F. Yamaguchi, "Leveraging flawed tutorials for seeding large-scale web vulnerability discovery," in *Proceedings of the 11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[77] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 519–529.

[78] S. A. Gorski III and W. Enck, "Arf: identifying re-delegation vulnerabilities in android system services," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, 2019, pp. 151–161.

[79] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 539–554.

[80] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.

[81] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[82] X. Shu, D. Yao, N. Ramakrishnan, and T. Jaeger, "Long-span program behavior modeling and attack detection," *Proceedings of the ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 4, pp. 1–28, 2017.