Information Sciences 553 (2021) 189–207

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Neural software vulnerability analysis using rich intermediate graph representations of programs

Seyed Mohammad Ghaffarian, Hamid Reza Shahriari [1],*

Department of Computer Engineering, Amirkabir University of Technology, Tehran, Iran

ARTICLE INFO

Article history:
Received 20 March 2020
Received in revised form 30 November 2020
Accepted 30 November 2020
Available online 15 December 2020

Keywords:
Automated vulnerability analysis
Software security vulnerability
Graph neural networks
Machine learning
Artificial neural networks

ABSTRACT

Security vulnerabilities are among the major concerns of modern software engineering. Successful results of machine learning techniques in various challenging applications have led to an emerging field of research to investigate the effectiveness of machine learning, and more recently, deep learning techniques, for the problem of software vulnerability analysis and discovery. In this paper, we explore the utilization of *Graph Neural Networks* as the latest trend and progress in the field of artificial neural networks. To this end, we propose an original neural vulnerability analysis approach, using customized intermediate graph representations of programs to train graph neural network models. Experimental results on a public suite of vulnerable programs show that the proposed approach is effective at the task of software vulnerability analysis. Additional empirical experiments answer complementary research questions about the proposed approach. In particular, we present experimental results for the challenging task of cross-project vulnerability analysis, with interesting insights on the capabilities of our novel approach. Furthermore, a software utility that was developed in the course of this study is also published as a further contribution to the research community.

1. Introduction

Computer software play a crucial role in the day-to-day lifestyle of modern society; ranging from simple apps on hand-held mobile devices, to sophisticated distributed enterprise software systems. A major concern in the modern software industry is the issue of security vulnerabilities that threaten the security and privacy of individual users, as well as high-value assets of public and private organizations. Various approaches have been proposed in the past two decades to overcome the issue of vulnerabilities in software engineering. Part of the past efforts and achievements are reviewed by Shahriar and Zulkernine [40]. Nevertheless, as some recent incidents[2] and latest industry studies[3], as well as academic studies [15] have shown, there is much room for progress with the current state of software security.

In addition to the traditional approaches reviewed in [40], an emerging paradigm is the utilization of *data science* and *artificial intelligence (AI)* techniques (specifically *machine learning*) to address the issue of software security vulnerabilities. An extensive survey of these approaches is presented by Ghaffarian and Shahriari in [14]. Some previous studies in this emerging paradigm have made promising contributions to the domain of software security (e.g. [48]).

* Corresponding author.
E-mail addresses: s.m.ghaffarian@aut.ac.ir (S.M. Ghaffarian), shahriari@aut.ac.ir (H.R. Shahriari).
[1] http://atlas.aut.ac.ir.
[2] https://www.wired.com/story/biggest-cybersecurity-crises-2019-so-far/.
[3] https://info.veracode.com/report-state-of-software-security-volume-10.html.

https://doi.org/10.1016/j.ins.2020.11.053
0020-0255/© 2020 Elsevier Inc. All rights reserved.

More recently, *deep learning* based on *deep neural networks* [24] have taken the spot-light of research efforts, due to their hugely successful results in various challenging applications; such as *image classification* [21], *machine translation* [42], *human face recognition* [43], and even *complex games* [41].

One of the latest achievements in the field of artificial neural networks is the introduction of *graph neural networks*. Traditional approaches in machine learning as well as modern deep learning, require tensor representations of data (e.g. vectors or matrices). Latest advances in artificial neural network research have enabled learning from complex structured data representations (e.g. graphs or networks). This new category of artificial neural networks has already shown promising results in some challenging application domains [20,25].

Graph representations are also popular in the domain of software or *program analysis*, where they are widely used to model computer programs in order to determine the existence of specific properties in computer programs [18,37,38]. There are various classic graphical representations for computer programs (e.g. Abstract Syntax Trees, Control Flow Graphs, Program Dependence Graphs, Call Graphs, etc.) that facilitate the analysis of computer programs in popular applications; such as compilers and integrated development environments (IDEs) [2].

In this original study, we aim to investigate the potency of recent advancements in graph neural networks in order to analyze and discover software vulnerabilities. To this end, we gathered a dataset of customized variants of well-known graphs generated from an industry-standard suite of vulnerable and secure programs, and experimented with modern graph neural networks to learn the task of distinguishing vulnerable programs from secure programs.

With this experimental setup, we pursued answers to several research questions: How effective are GNN models for the task of program vulnerability analysis? How various graph-representations affect the performance of GNN models? Our results show that the proposed graph-based neural vulnerability analysis system outperforms baseline methods by a significant margin and proves to be effective for the task at hand.

Furthermore, we conduct experiments for the challenging task of cross-project vulnerability analysis. The results show that the proposed approach presents prime opportunities for an acceptable solution to this challenging task. We also discuss some interesting insights gained from the experimental results.

Other than the aforementioned proposed approach, we developed a tool named PROGEX[4] in the course of this study which enabled us to extract various graph representations of programs in a suitable format to be used in the experiments. We have released this helpful tool with a liberal open-source license as an additional contribution to the research community.

The rest of this paper is organized as follows: in Section 2 we first present brief background information about the problem domain and the key techniques that we utilize in this study, followed by a review of previous related works. In Section 3 we begin with an overview of the proposed approach, followed by more elaborate descriptions of key components. In Section 4 we describe our experimental setup, evaluation methodology, as well as some implementation details. In Section 5 we present experimental results, discussions, and insights about each research question. Section 6 concludes the paper with some remarks and future work.

2. Background and related work

2.1. Software vulnerability analysis

The definition of *software security vulnerability* is stated by Ghaffarian as:

> " *A software vulnerability is an instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy.* " [14].

Program vulnerability analysis is the process to decide whether a given program contains known security vulnerabilities or not. The output of a vulnerability analysis system is to approve or disapprove the security of a given program (i.e. binary output); whereas, a more practically useful system is a *program vulnerability discovery* (or *vulnerability reporting*) system, which outputs more detailed information (such as type, precise location, possible fixes, etc.) for each vulnerability discovered in a given program. It has been shown that the problem of software vulnerability analysis is undecidable in the general case. What undecidability means to the practitioner is that a sound and complete solution to the problem does not exist; that is, a software vulnerability analysis or discovery system which guaranties to detect no false vulnerabilities, and not miss any actual vulnerabilities, is known to be non-existent [14,22].

Nevertheless, due to the crucial importance of the matter of software security vulnerabilities, both the academic community and the software industry have proposed a plethora of (inevitably approximate) mitigation techniques to tackle the problem in the past two decades [5]. The majority of previous approaches have roots in the field of *program analysis* [34], and an extensive review of previous work is presented in [40].

In addition to traditional techniques, there is a modern class of works that utilize techniques from the fields of data science and artificial intelligence (AI) to address the problem of software vulnerability analysis and discovery. The advancements of machine learning algorithms in recent years, and the success stories of utilizing data mining to address many difficult application problems have motivated researchers to more thoroughly investigate the effective utilization of these

[4] https://github.com/ghaffarian/progex/.

techniques for difficult problems in the domain of computer security and privacy. An extensive survey of such studies on the problem of software vulnerability analysis and discovery is presented in [14].

In a position paper (yet to appear in peer-reviewed publication), Habib and Pradel [16] present the concept of *neural bug finding*. The authors argue that while the use of static analysis is widely adopted in the software industry, developing effective static analysis tools requires high expertise that is challenging to acquire. They propose formulating static analysis problems as a classification problem using deep neural networks, as an alternative approach to developing effective program analysis tools. It is noteworthy that the idea of using machine learning for bug-finding and other challenging program analysis applications is previously studied by many research papers, as reviewed and discussed by Ghaffarian and Shahriari [14].

2.2. Graph neural networks

Graphs are rich and well-established representations to model computer programs, facilitating various types of analysis [37]. Yet, machine learning on graphs is relatively less-regarded compared to prominent machine learning algorithms which mainly process data in tensor form (vectors, matrices, etc) [14]. While graphs are powerful representations to model relationships among data objects, this comes at the cost of greater computational complexity for many basic tasks in pattern recognition; such as measuring similarities for graph matching [13,30].

One of the more recent successful approaches to machine learning on graphs is *graph neural networks*, which avoids many combinatorial complexities of processing graphs. The first variants of graph neural networks were proposed by Scarselli [39] but had several limitations in both training speed and learning capability, which greatly hindered their applicability. In recent years, with inspiration from techniques in deep neural networks, several improved variants of graph neural networks were proposed by researchers (e.g. [20,25,44]). Modern GNNs present much favorable inductive bias relative to previous neural networks [8].

A position paper by a group of researchers from Google Brain and DeepMind presents a unification model (named *graph networks*) for almost all variants of graph neural networks. They propose that *combinatorial generalization* and *relational reasoning* should be a top priority for the future of AI systems, which *graph neural networks* are a key to its realization [6]. There are two (yet to appear in peer-reviewed publication) survey papers, that cover different variants of graph neural networks and discuss challenges as well as shortcomings of each approach [45,49]. Furthermore, some recent papers provide a theoretical foundation and formal mathematical analysis on the capabilities of graph neural networks [11,46].

In conclusion, *graph neural networks* are a new and young method introduced in the field of artificial neural networks, facilitating knowledge acquisition from the rich world of complex structured data representations. There is good agreement among the research community that this new learning method has a lot of potential for the future of AI [6], and a promising research direction for application domains with structured data representations (such as *Program Analysis*). This is one of the main motivations for the current study at hand.

2.3. Related work

In this section, we review some of the most relevant previously published work which have utilized data mining or machine learning techniques for software vulnerability analysis and discovery, in chronological order. For the sake of completeness, we also review some related work from the field of *Software Fault Localization* which have utilized relevant techniques [12].

2.3.1. Graph mining for software fault localization

Chang et al. [9] investigated the use of graph-mining and matching techniques in an anomaly-detection fashion to discover *missing-check defects* in a given program. Experimental results on four open-source projects show high false positive rates by the proposed anomaly-detection approach [9].

Cheng et al. [10] propose a top-k discriminative subgraph mining algorithm by extending a previous frequent subgraph mining algorithm named LEAP. The proposed algorithm is used to extract discriminating subgraphs from execution graphs extracted at runtime to distinguish correct and faulty execution traces. Results on small benchmark programs show minor improvement compared to previous studies who employ sequence mining techniques [10].

Parsa et al. [35] propose another top-k discriminative subgraph mining algorithm by improving a previous work by the same authors. The proposed algorithm is used to distinguish correct or faulty execution traces. Execution graphs are extracted at runtime by program instrumentation. The mining algorithm employs edge-weights for improved mining results and a pruning strategy to improve runtime performance. Experimental results on small benchmark programs show minor improvements compared to [10].

At their core, most classic combinatorial techniques of *graph-mining* reduce to the *subgraph isomorphism problem* which is known to be NP-complete; or the *maximum common subgraph problem* which is known to be NP-hard [30]. Because of the severe complexity order, such techniques cannot scale to software projects with a moderate to large size, nor using full-featured graph representations of programs, containing rich attributes thus far [12].

2.3.2. Deep learning for program analysis

Pradel and Sen [36] present a machine-learning approach to tackle the problem of name-based software bugs. The authors formulate the problem as a binary classification and perform experiments on a large corpus of JavaScript files. The learning data is generated by performing simple transformations on sample code to introduce name-based bugs. Code samples are pre-processed to extract identifier names from expressions and using a variant of the well-known Word2Vec model, distributed tensor representations are generated for each code sample. This training data is fed to a shallow feed-forward neural network for training and evaluating the classification model [36].

Li et al. [28] propose the first vulnerability detection system for C/C++ programs, using deep-learning techniques. The proposed approach uses a large corpus of synthetic vulnerable programs (limited to only two types of vulnerabilities) to train a BLSTM deep neural network for classifying programs. The input data is generated by a preprocessing phase in which program slices dependent on specific API calls are extracted as code gadgets. These code gadgets are then tokenized and vectorized via the well-known Word2Vec technique [28]. The same authors have also presented extended versions of VulDeePecker which are yet to appear in peer-reviewed publications; including a multi-class detection variant named *μVulDeePecker* [50], and a more general detection framework for C/C++ programs, named *SySeVR* [27].

Yahav et al. [47] present preliminary experimental results on employing deep neural networks for two programming languages tasks. The authors present an embedding technique named *code2vec* that is used for the task of *code snippet captioning*. They also present a sequence embedding technique named *code2seq* that is used for the task of *code completion* [47].

Li et al. [26] continue their previous study on training deep neural networks for detecting vulnerabilities in C/C++ programs, by extending the experiments to answer some research questions for more insight. First, experimental results show that incorporating more semantic information in input data yields performance improvements for the BLSTM neural network. Second, employing data balancing techniques had a negative effect on classification performance of their particular dataset. And third, bidirectional deep neural networks have minor advantage in terms of classification performance compared to unidirectional networks [26].

Liang et al. [29] propose a bug localization system named CAST which is designed to assist in resolving issues by ranking program source files based on related bug reports. For this purpose bug reports as well as program source files are tokenized and vectorized using the well-known Word2Vec technique and fed to a deep convolutional neural network (CNN) for training. Moreover, customized Abstract Syntax Trees are extracted from source files and fed to a Tree-based CNN. The outputs of all the aforementioned networks are fused by a fully connected (dense) neural network to train the final classifier. Experimental results show minor improvement compared to previous state-of-the-art work [29].

Jana and Opera [19] propose an unsupervised anomaly detection system named AppMine for attack/exploitation-detection in containerized web applications. AppMine employs a black-box approach; that is no program source code is analyzed, but only application activity data is collected using a monitoring tool which gathers the sequence of system calls made by the application under test inside the container. The input vector at any given time period is constructed from the frequency of various system calls in that period. For model training, only legitimate program behavior is used to train the unsupervised model. The popular LSTM deep recurrent neural network is compared to two traditional models for unsupervised machine learning (PCA, and one-class SVM). Experimental results show the superiority of the LSTM compared to the other models [19].

2.3.3. Graph neural networks for program analysis

Mou et al. [33] present a variant of graph neural networks named *Tree-based Convolutional Neural Network (TBCNN)*, suited for tree-structures as input data. The proposed GNN consists of a convolution-layer using a *tree-based convolution kernel* proposed by the authors, and a *dynamic pooling layer* to extract the most important information from any input tree. This data is then fed to a fully-connected Feed-Forward Neural-Network, consisting of a single hidden-layer and an output-layer. Abstract Syntax Trees (AST) are one of the most prominent intermediate representations in program analysis applications [2]. The authors evaluate the effectiveness of the proposed neural network by experiments on two software engineering tasks that take ASTs as input. The authors also make comparisons with some deep neural networks as well as some traditional neural networks, where the proposed TBCNN achieves much better accuracy [33].

Li et al. [25] present two variants of Graph Neural Networks named *Gated Graph Neural Network (GGNN)*, and *Gated Graph Sequence Neural Network (GGSNN)*, where both models include an adaptation of the propagation-model inspired by recurrent-units of modern Deep Neural Networks. These adaptations enable the network to learn some embedding for the entire graph in the GGNN case; and in the GGSNN case, it also enables learning intermediate embeddings for each node when producing a sequence of outputs. The proposed models are evaluated with several toy problems of graph theory; which shows that GGSNN is able to achieve better accuracy than RNN and LSTM (two well-known deep models) with five-times fewer training samples. The authors also propose an approach for the task of *program verification*, where a formulation using GGSNN is presented to predict the property of *memory safety* for a given *heap objects diagram* as a graph. Experimental results show the effectiveness of this approach compared to some existing complex solutions [25].

Allamanis et al. [4] propose an unsupervised learning approach using *Gated Graph Neural Networks (GGNN)* on a large corpus of open-source C# programs as a solution for two software engineering tasks: variable naming and variable misuse detection. The authors propose a customized AST representation which includes `NextToken` edges between syntax-tokens at the tree-leaves, as well as data-flow edges among the syntax-tokens. This enriched AST is fed to a GGNN to train the model for a fixed number of steps. The authors present some quantitative as well as qualitative evaluation results to show the effectiveness of their proposed approach. Quantitative results show %66 F1-score for the within-project setup, and %62 F1-score for the cross-project setup [4].

Brockschmidt et al. [7] present another similar unsupervised approach using an *asynchronous graph neural network*. The authors investigate the idea of *neural attribute grammars*, where attributes in the nodes of a program's abstract syntax tree (AST) are computed from both parent and child nodes using the aforementioned GNN. Evaluation was done for the problem of code generation, with a *fill-in-the-blank* experimental setup [7].

3. Proposed approach

3.1. Hypotheses

As described in Section 2.1, the problem that we address in this study is *program vulnerability analysis*, which is a system that takes a program as input, performs its analysis, and outputs a decision about the vulnerability status of the given program (either `vulnerable` or `secure`). We aim to use a modern machine learning approach for this problem, but unlike previous studies that transform programs to flat tensor representations suitable for major machine learning algorithms, we hypothesize that using rich graph representations of programs can yield superior results:

Hypothesis **3.1.** Using rich graph representations of programs for the task of learning program vulnerability analysis, can yield superior results compared to flat vector representations of programs.

Yet as discussed in Section 2.2, performing machine learning on rich graph representations is a highly challenging task to accomplish. With the advent of graph neural networks, prime opportunities arise in various application domains. Hence, we put forth our main hypothesis:

Hypothesis **3.2.** Using rich graph representations of programs to train a model based on modern graph neural networks can yield an effective solution for the task of program vulnerability analysis.

To verify the aforementioned hypotheses, an experimental design for a *neural vulnerability analysis system*, is proposed that automatically learns to decide about the vulnerability status of input programs using a rather moderate set of sample programs. Our proposed approach for this problem is to use *graph neural networks* that are trained using rich graph representations of sample programs.

3.2. Approach overview

An overview of the proposed approach is presented in Figure 1. The proposed approach is a supervised learning setup, consisting of two phases: a *training phase* which infers the neural vulnerability analysis model given a corpus of example programs, and a *utilization phase* which the inferred model is used to analyze new programs whose vulnerability status is unknown.

The training phase starts by gathering a corpus of sample programs for each desired security vulnerability to be analyzed by the learning system. For model training in a supervised setup, the input corpus is required to have labels; that is, the vulnerability status of each sample program is required to be known beforehand. Gathering such corpus is certainly a challenge, and our decision for this study is explained in Section 4.1.

After gathering the training corpus, each sample program should be represented in some rich graph form to contain enough information required for the purpose of vulnerability analysis. According to [48] both syntax and semantic information (including control-flow and data-dependency) of programs is required for effective vulnerability analysis. Extracting such graphs from programs in a format suitable to be used in machine learning systems is another challenge and in Sections 3.3 and 4.2 we present our solution.

Mainstream machine learning methods require numeric tensor data (such as vectors and matrices) in order to perform their calculations and construct models. This is also true for modern deep neural networks. While graph neural networks enable us to analyze data in graph form, the vertex/edge attributes of the graphs are still required to be numeric tensors. Our customized variants of intermediate graph representations of programs contain attributes in textual format (e.g. variable identifiers, method names, object types, language keywords, etc). Effective transformation of

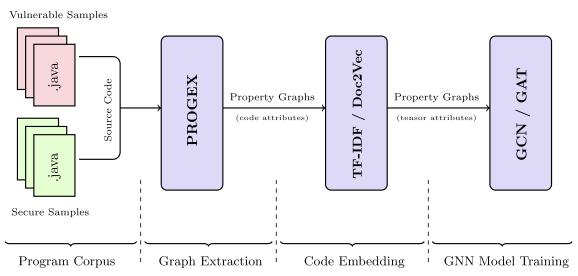

Fig. 1. Overview of the proposed approach (training phase).

textual attributes in the graph's vertices/edges to numeric tensor attributes is another challenge, and in Section 3.4 we present our solution.

After overcoming all the aforementioned challenges, the data is ready to be fed to graph neural networks. In our proposed approach, two modern GNNs are trained to construct neural vulnerability analysis models and evaluate their effectiveness. In the next sections, we will cover more details about various components of the proposed approach.

3.3. Graph representations of programs

Graphs are a fairly established data model in the field of program analysis [18,37,2]. For constructing intermediate graph representations of programs, a given program is analyzed by a respective parser of the target programming language. The result of the parsing process is a *parse-tree* for the input program. Parse-trees are used by compilers and other program analysis applications, but they contain too much detailed information for each element of the program which makes them crowded and too large. The *abstract syntax tree* (AST) is a directed tree, which acts as a more concise version of the parse-tree by abstracting some of the details.

Another useful and fairly popular intermediate graph representation is the *control flow graph* (CFG) which is a digraph where vertices are program statements and a directed edge from a statement s_a to another statement s_b denotes the order of execution; $s_a \rightarrow s_b$ means the execution of statement s_a is followed by the execution of statement s_b. The set of possible edge labels are $\{\epsilon, \mathit{true}, \mathit{false}\}$ which denote the control-flow conditions. The CFG is constructed by analyzing the AST or the program parse-tree and has two variations: inter-procedural and intra-procedural.

The program dependence graph (PDG) actually consists of two subgraphs: control dependence graph (CDG) and data dependence graph (DDG) [18]. The CDG is a directed tree where vertices are program statements, and each vertex has a control dependency on its parent vertex. If statement s_a is control dependent on statement s_b, it means that the evaluation of s_b can decide whether s_a is executed or not. In this case, an edge $s_b \rightarrow s_a$ is present in the CDG. The CDG is constructed by analyzing the AST or the program parse-tree.

The DDG is a multi-digraph where vertices are program statements, and directed edges denote a data-flow from the source statement to the destination statement. The construction of the DDG requires both intra-procedural control-flow information, as well as variable definition/usage (or *Def-Use*) analysis for all program statements.

In this study, we aim to experiment with customized variants of the aforementioned graph representations, including Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). Our desired model for rich intermediate graphs is in the form of *property graphs*. The formal definition of a property graph is as follows:

Definition 3.1. *[Property Graph].* A *property graph* $g = (V, E, \Lambda, \Sigma, \mu, \lambda, \sigma)$ is a directed, attributed, multi-graph; where V is the set of vertices (or nodes) of g, E is the set of edges (or links) of g, $\mu : E \rightarrow V \times V$ is the adjacency function of the graph which maps any given edge to its ordered pair of source and destination vertices, $\lambda : V \rightarrow \Lambda$ is the attribute function for vertices which maps any given vertex to its attributes, and $\sigma : E \rightarrow \Sigma$ is the attribute function for edges which maps any given edge to its attributes.

Listing 1: Basic Java code example

```
public class Basic {

  private boolean isEven(int num) {
    int mod = num % 2;
    return mod == 0;
  }

  public void run() {
    int[] tests = {1, 2, 3, 4, 5};
    for (int i: tests) {
      if (isEven(i))
        System.out.println(i + " is even");
      else
        System.out.println(i + " is odd");
    }
  }
}
```

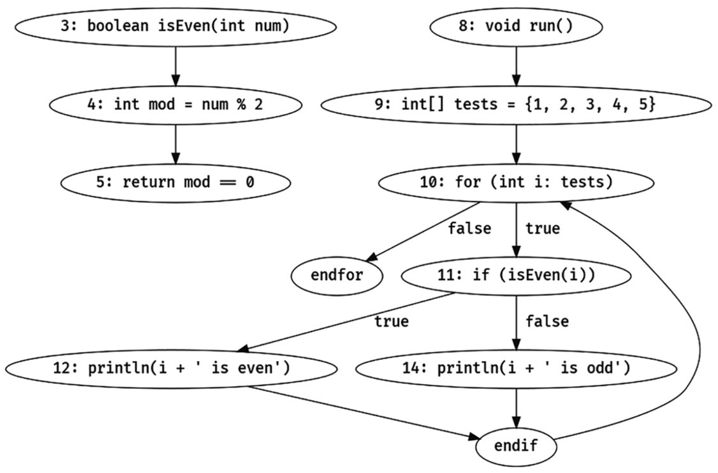

Fig. 2. Control Flow Graph (CFG) of basic Java example.

As an example, consider the basic Java program in Listing-1. The corresponding customized CFG for the code in Listing-1 is shown in Fig. 2. The customized CFG in Fig. 2 is an example of our desired property graph; which the set of possible edge-attributes is $\{\epsilon, true, false\}$, and vertices contain multiple attributes, such as `line`, `code`, etc.

3.4. Tensor attributes for graph vertices

The property-graph representations extracted from programs can contain attributes of various types. For example, vertices of the CFG of Fig. 2 have a `line` attribute of type `integer`, and a `code` attribute containing program expressions or statements in textual form (of type `string`). Such data is not suitable for processing using graph neural networks, and instead should be converted to numerical tensors.

There are several well-known approaches for transforming textual data into numerical tensors that have been used in previous studies. One of the popular traditional approaches is the *Term-Frequency Inverse-Document-Frequency* or TF-IDF technique from the field of *information retrieval* [31]. Simply explained, TF-IDF is an example of *latent semantic analysis* (LSA) techniques that given a corpus of documents, provides a score for each term in every document of the corpus. Using TF-IDF, the score of term t in a document d from a corpus D is calculated as:

$$\mathbf{TF} - \mathbf{IDF}(t, d, D) = f_{t,d} \times \log \frac{|D|}{|D_t|} \tag{1}$$

where $f_{t,d}$ is the frequency of term t in document d; $|D|$ is the number of documents in the corpus; and $|D_t|$ is the number of documents in D which contain term t. After calculating the score for every term in all documents, any document can be transformed into a numerical tensor by representing the document as a vector consisting of the scores of all terms in that document. While the TF-IDF technique is widely used with many successes, it has various limitations [31].

One important limitation of TF-IDF is the high dimensionality of tensors produced by this technique. One remedy for this issue is to use a statistical dimensionality reduction technique, such as *Principal Component Analysis* (PCA) or *Singular Value Decomposition* (SVD). Such approaches perform an orthogonal linear transformation and are able to project high dimensional data into a lower dimension space [1].

More recently, along the successes of modern neural networks, a new class of techniques known as *distributed word embeddings* was proposed in the field of *natural language processing* (NLP), that have been adopted with excellent results. The most widely known distributed word embedding approach is the popular *Word2Vec* technique [32].

Word2Vec is a shallow neural network model that is trained to map individual words from a given text corpus to a target tensor space, such that the resulting tensors for words that are used in similar context have close proximity in the target space [32]. *Doc2Vec* is an extension of Word2Vec, that can map sentences, paragraphs, or even full length documents to some target tensor space [23].

Using any of the aforementioned techniques, one can transform textual vertex attributes of property graphs into numerical tensors suitable to be processed using GNNs. In this study, we have experimented with variants of both types of techniques for this purpose (Section 5.3).

As an example, suppose we transform all code attributes of the CFG in Fig. 2, using some embedding method such as TF-IDF with SVD to generate numeric tensor attributes of size 64. Edge attributes are also converted to a single value vector. Fig. 3 shows the CFG of the example program in Listing-1, after transforming node and edge attributes to numeric tensors.

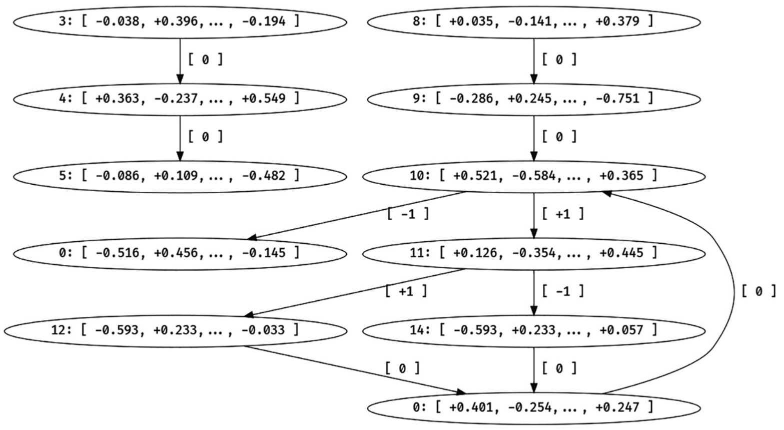

Fig. 3. CFG of basic Java example with numerical tensor embeddings for nodes & edges.

3.5. GNN models

As discussed in Section 2.2, several refined variations of graph neural networks were proposed in the literature; including: *Graph Convolutional Networks* (GCN) [20], and *Graph Attention Networks* (GAT) [44]. At their core, both models perform some calculations at each layer which can be simplified as below:

$$H^{(l+1)} = f\left(H^{(l)}, A\right) \tag{2}$$

where $H^{(l)}$ is the feature matrix for the lth layer of the GNN, with $H^{(0)}$ being the input matrix constructed from initial vertex attributes; A is the adjacency matrix of the input graph, and $f(\cdot)$ is some non-linear function. The output feature matrix at the last layer of the GNN consists of output representations for every vertex of the graph, which can be used for vertex-level predictions. For graph-level predictions, some form of pooling operation (such as summation, average, etc.) is used.

For Graph Convolutional Networks (GCN) [20], a convolution operation aggregates the normalized sum of vertex features in the single-hop neighborhood for all vertices. This yields the following equation; where h_i^l is the feature tensor for the ith unit of the lth layer of the GCN, $\mathcal{N}(i)$ is the set of single-hop neighbors for ith unit in this layer, c_{ij} is a normalization factor, $W^{(l)}$ is a weight matrix for units of the lth layer of the GCN, and $\sigma(\cdot)$ is the ReLU[5] non-linear activation function:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right), \quad c_{ij} = \sqrt{|\mathcal{N}(i)|}\sqrt{|\mathcal{N}(j)|} \tag{3}$$

Kipf and Welling [20] propose two normalization techniques for the GCN propagation equation to overcome some issues in feature scaling and learning stability. The authors' final proposed propagation equation for GCN is as follows; where $\widehat{A} = A + I$, and $\widehat{D}$ is the diagonal vertex degree matrix of $\widehat{A}$:

$$H^{(l+1)} = \sigma\left(AH^{(l)}W^{(l)}\right) = \sigma\left(\widehat{D}^{-\frac{1}{2}}\widehat{A}\widehat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{4}$$

Fig. 4 shows an example of a basic Graph Neural Network architecture using the CFG of Fig. 3 as an input graph. A GCN layer at the input performs the message-passing operation (as defined in Eq. 3) among all vertices of the input graph, where each vertex computes a new embedding based on the embeddings of all neighboring vertices with some learnable weight matrix [6]. If the network has more GCN layers, then the newly computed embeddings are passed to the next layer (as in Eq. 4); otherwise, the newly computed embeddings are aggregated using some aggregation function (summation, average, etc.), which computes a single embedding for the entire graph, and passes it to the next layer of the network, which is the input of a well-known *Multi-Layer-Perceptron* (MLP) network (also known as *Feed-Forward Neural-Network*). The MLP computes the output values, and the outputs are presented to a `softmax` function to determine the output label. In the training phase, this output is used as feedback with the well-known *Back-Propagation* algorithm to tune the weights all the way back to the input GCN layer.

[5] Rectified Linear Unit.

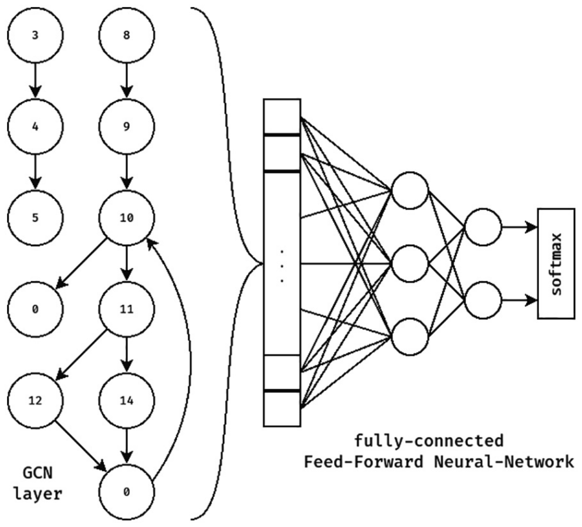

Fig. 4. Example of a basic Graph Neural Network (GNN) with the CFG of Fig. 3 at its input. This example GNN uses a single GCN layer, and a fully-connected Feed-Forward Neural-Network (FFNN) with a single hidden layer of 3 units, and one output layer of 2 units. As specified in Section 3.5, a graph embedding is computed in the GCN layer, which is fed to the FFNN as the input vector.

The *Graph Attention Networks* (GAT) [44] differs from GCN by introducing an attention score between units of the GNN. The propagation equation of GAT is very similar to that of GCN, differing by replacing the normalization factor with the attention mechanism:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)}\right) \tag{5}$$

$$\alpha_{ij}^{(l)} = \text{softmax}_{(i)}\left(e_{ij}^{(l)}\right) = \frac{\exp\left(e_{ij}^{(l)}\right)}{\sum_{k\in\mathcal{N}(i)} \exp\left(e_{ik}^{(l)}\right)} \tag{6}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}\left(\vec{a}^{(l)^T} \cdot \left[W^{(l)} h_i^{(l)} || W^{(l)} h_j^{(l)}\right]\right) \tag{7}$$

where $\alpha_{ij}^{(l)}$ is the normalized attention score between unit i from the lth layer and its jth neighboring unit, computed by applying a `softmax` on un-normalized pair-wise attention score e_{ij}, and e_{ij} is computed by first concatenating the linear transformations of previous feature embeddings $[W \cdot h_i || W \cdot h_j]$ and then computing its dot product with the transpose of a learnable weight vector $\vec{a}$, and finally applying the `LeakyReLU` non-linear activation function.

The multi-head additive attention mechanism introduced in GAT enhances the capacity and expressiveness of the model compared to GCN while also makes the training process more time-consuming, and training convergence becomes more challenging. Experimental results by Velickovic, et al. [44] shows that in some cases the performance of GAT is on par with GCN, while in some other cases GAT shows some advantages.

The architectures used in this study are similar to the architectures presented by Kipf & Welling [20], and Velickovic et al. [44]. For the GCN model, we use 3 layers of GCN: one input, one hidden, and a final output layer. Similarly, for the GAT model, we use 3 layers of GAT: one input, one hidden, and one final output layer. In both models, the final embeddings of the output layers are aggregated using an element-wise summation and presented to a `softmax` function to determine the network's decision.

3.6. Research questions

In accordance with the scientific method [17], we perform some empirical experiments to validate our hypotheses from Section 3.1 and evaluate the efficacy of the proposed approach. To this end, we aim to provide answers to several research questions as follows:

- *RQ-1:* How effective is the proposed approach compared to various baseline methods for the task of software vulnerability analysis?

- *RQ-2:* How various graph-representations of programs affect the performance of the proposed approach for the task of software vulnerability analysis?
- *RQ-3:* How various embedding of code-attributes affect the performance of the proposed approach for the task of software vulnerability analysis?
- *RQ-4:* How does code normalization affect the performance of the proposed approach for the task of software vulnerability analysis?
- *RQ-5:* How effective is the generalization capabilities of the trained GNN models for the task of cross-project software vulnerability analysis?

The first research question is the falsification of our hypotheses. The rest of the research questions examine the effect of various design choices, in order to present a complete research study. The final research question evaluates the generalization capabilities of the proposed approach in the challenging cross-project setup.

4. Empirical evaluation

In this section, we present our empirical experimental setup, implementation details, and evaluation metrics used to answer the desired research questions of Section 3.6.

4.1. Experimental setup

Our empirical evaluation method is to use a controlled dataset consisting of a balanced number of vulnerable and secure programs in order to train and evaluate modern graph neural network models for the problem of program vulnerability analysis. In the following, we discuss the rationale behind key decisions in our experimental setup design.

4.1.1. Target programming language

In this study, we decided to focus on *Java* as the programming language of choice in our experiments. By various industry metrics, Java has been consistently among the top three programming languages of the past decade (IEEE Spectrum ranking,[6] GitHub's PYPL,[7] and TIOBE's index[8]). This clearly shows the importance of the Java language in the software industry, and hence makes it a reasonable choice to focus our research efforts.

4.1.2. Target software vulnerabilities

In this study, we decided to focus on *web application vulnerabilities*. According to the latest edition of MITRE's *Top 25 Most Dangerous Software Vulnerabilities*[9] published in 2019, only six out of the top 25 vulnerabilities are exclusively unrelated to web applications. This clearly shows the importance of web application security. On the other hand, the Java programming language is one of the most prominent choices for enterprise, large-scale, and mission-critical web applications.[10]

4.1.3. Supervised vs unsupervised

As reviewed in Section 2.3, some previous studies have experimented on the effectiveness of modern graph neural networks in an unsupervised setup for various program analysis tasks (although none have explored the problem of software vulnerability analysis). In this study, we aim to experiment the utilization of modern graph neural networks in a supervised fashion and explore its effectiveness for the problem of software vulnerability analysis. On the other hand, a supervised setup has more challenges that need to be addressed in a research study.

4.1.4. Vulnerability dataset

Since we are aiming for a supervised setup in this study, we require a balanced and labeled set of programs whose vulnerability status is known a priori. There are very few labeled datasets for web application vulnerabilities that are suitable for a supervised machine learning study. Based on our analysis, the most suitable overall choice was the OWASP Benchmark suite.[11] The OWASP Benchmark project is a synthetic Java test suite designed for the automated evaluation of software vulnerability analysis tools. At the time of this study, the OWASP Benchmark test suite has two main versions (v1.1 and v1.2), consisting of thousands of labeled test cases in 11 categories for web application vulnerabilities. In this study, we selected 7 categories of the OWASP Benchmark suite to conduct our experiments, whose details are presented in Table 1. The other categories that we left out of this study were simple programming mistakes related to utilizing cryptography APIs.

Version 1.1 of the OWASP Benchmark has a total of 14,124 samples in all the 7 selected categories of Table 1, while v1.2 has a total of 1,698 samples, which is almost 8 times fewer; but the test suite is not executable, nor exploitable, and the vul-

[6] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019.
[7] http://pypl.github.io/PYPL.html.
[8] https://www.tiobe.com/tiobe-index/.
[9] https://cwe.mitre.org/top25/.
[10] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019.
[11] https://owasp.org/www-project-benchmark/.

Table 1
Details of selected categories from the OWASP Benchmark suite (v1.1 and v1.2).

CWE	Title	Ver	Test-Cases	Vulnerable	Secure
22	Path Traversal	v1.1	2,630	1,706	924
		v1.2	268	133	135
78	Command Injection (CMDI)	v1.1	2,708	1,802	906
		v1.2	251	126	125
79	Cross-Site Scripting (XSS)	v1.1	3,449	1,540	1,909
		v1.2	455	246	209
89	SQL Injection (SQLI)	v1.1	3,529	2,297	1,232
		v1.2	504	272	232
90	LDAP Injection (LDAPI)	v1.1	736	521	215
		v1.2	59	27	32
501	Trust Boundary Violation	v1.1	725	505	220
		v1.2	126	83	43
643	XPATH Injection	v1.1	347	217	130
		v1.2	35	15	20

nerable and secure samples are not balanced. Version 1.2 of the test suite improves in all these aspects: the test suite is fully executable, the vulnerable samples are exploitable, and the samples are balanced. In this regard, OWASP Benchmark v1.2 is a more realistic code-base for research, and the balanced samples are an important feature for machine-learning algorithms; yet the fewer number of samples may present some challenges.

4.1.5. Experiments platform

All the experiments in this study were executed on a single local computer with a 64-bit Ubuntu Linux (18.04 LTS) operating system, 12 gigabytes of RAM, and an Intel Core i5-8400 processor with 6 cores. No graphics processing unit (GPU) was utilized in this study.

4.2. Implementation

The various tools and technologies that were used in this study to implement the experiments are as follow:

4.2.1. PROGEX: program graph extractor

Since we were unable to find a suitable software tool to extract our desired intermediate graph representations in a standard reusable format by external tools and libraries, we had to develop such a tool on our own. The result of our efforts is a cross-platform tool named PROGEX,[12] with the ability to extract various graph representations from program source code. PROGEX was designed and implemented based on the capable ANTLRv4[13] parser generator. According to our experiments, PROGEX is able to efficiently analyze code-bases and extract graphs such as AST, CFG, and PDG with high speed. PROGEX is also able to export the resulting graphs into popular file formats for graphs such as DOT, JSON, and GML; which can be used both for visualization and analysis by external tools and libraries. We have released PROGEX under a liberal open-source license, as a contribution to the research community.

4.2.2. Python machine learning libraries

The main programming language that we used to implement our experimental system is the Python language and its plethora of libraries for data analysis and machine learning. Some of the key libraries that we used in our implementations are the Scikit-Learn[14] machine learning library, the NetwokX[15] library for graph analysis, the Gensim[16] library for latent semantic analysis of text corpora, and the PyTorch[17] deep learning library.

4.2.3. DGL: deep graph library

One key library which facilitated our experiments for this study is the Deep Graph Library (DGL),[18] which is another Python library based on PyTorch that provides several utilities to facilitate the implementation and efficient execution of modern graph neural network models. Although DGL is still in the early phases of development and yet far from reaching a stable version 1.0, we acknowledge that it was a valuable help for the realization of this study.

12 https://github.com/ghaffarian/progex/.
13 https://www.antlr.org/.
14 https://scikit-learn.org/stable/.
15 https://networkx.github.io/.
16 https://radimrehurek.com/gensim/.
17 https://pytorch.org/.
18 https://www.dgl.ai/.

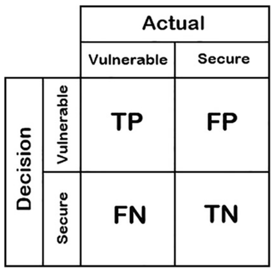

Fig. 5. Confusion matrix for the output decision of a vulnerability analysis system, against the actual vulnerability state of each input program. (TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative).

4.3. Evaluation metric

For the purpose of model evaluation, we utilize the well-known *F1-score* metric which is fairly established in the machine learning literature. The formula for the F1-score metric is presented based on the parameters of the confusion matrix in Fig. 5, which interprets the output decision of a vulnerability analysis system for any given input program.

$$\textbf{Precision} = \frac{TP}{TP+FP}, \quad \textbf{Recall} = \frac{TP}{TP+FN} \tag{8}$$

$$\textbf{F1}-\textbf{score} = 2 \times \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{9}$$

5. Results and discussion

In this section, we present the experimental results for each research question and discuss the results for insights. For the purpose of evaluation, we calculate the F1-score in all experiments as defined in Section 4.3. For machine learning models, we perform 5-fold cross-validation to calculate and report the F1-score; that is, the input data is randomly partitioned into five equal folds and each model is trained using four partitions and evaluated with the fifth partition. This is repeated five times, where each time one of the five partitions is used for evaluation. Finally, the mean value of all five evaluations is used as the final F1-score for that model.

5.1. Proposed approach vs baseline methods

RQ-1: **How effective is the proposed approach compared to various baseline methods for the task of software vulnerability analysis?**

Our first research question aims to see if the proposed *neural vulnerability analysis systems* based on *graph neural networks (GNNs)* is effective for the task at hand. To answer this, we shall compare the performance of our selected GNN models against some effective baseline methods for the same task. Baseline methods consist of both well-known static analysis tools, as well as effective machine learning techniques from the literature.

For static analysis tools, we aimed for one of the most well-established open-source static analysis tools in the Java industry which has detection rules for web security vulnerabilities: SpotBugs[19] (with the Find-Security-Bugs[20] plugin). According to a recent study [15], SpotBugs was the most effective static analysis tool compared to some other open-source tools. On the other hand, the Find-Security-Bugs plugin for SpotBugs recently became an OWASP Foundation project.[21] For the best possible results, we have used the latest version of Find-Security-Bugs with the following command parameters:

```
$./findsecbugs.sh -high -progress -sortByClass -xml -output report.xml \
  -onlyAnalyze org.owasp.benchmark.testcode.* owasp-benchmark.war
```

For the baseline machine learning method, we selected the most effective shallow-learning model in previous studies; which according to our survey [14] is the well-known Random-Forest technique. Random-Forest is an ensemble learning technique that has proven highly effective in practice [14]. For training the baseline shallow-learning model, we apply

[19] https://spotbugs.github.io/.
[20] https://find-sec-bugs.github.io/.
[21] https://owasp.org/www-project-find-security-bugs/.

Table 2
F1-score results for RQ-1 experiments.

Method	CWE-22	CWE-78	CWE-79	CWE-89	CWE-90	CWE-501	CWE-643
FSB	.5533	.5972	.6236	.6885	.5085	.3540	.5854
RFT	.7585	.7618	.7385	.7523	.7121	.7142	.6571
GCN	.9123	.9259	.8899	.9476	.7381	.9044	.6762
GAT	.7788	.8137	.8036	.8861	.6167	.8711	.5812

FSB: SpotBugs with Find-Security-Bugs plugin.
RFT: Random-Forest model.
GCN: Graph-Convolutional-Networks model.
GAT: Graph-Attention-Networks model.

the TF-IDF method with SVD analysis to transform the sample programs corpus into vector representations that are usable for training the selected baseline model. The feature vector size was set to 256 for this experiment.

For the proposed models, we use *control-flow graphs* (CFG) to train and evaluate two GNN models: *Graph Convolutional Networks* (GCN) [20] and *Graph Attention Networks* (GAT) [44]. The same TF-IDF with the SVD method was applied as the code embedding technique used for vertex attributes of the input graphs to make the experiments invariant to unrelated factors and mainly measure the effect of using graph models compared to other models. The results are presented in Table 2.

According to our experiments, the latest version of Find-Security-Bugs (v1.10.1 published Oct 2019) performed considerably better than what is reported on the OWASP Benchmark website. While this can be due to the latest developments by the Find-Security-Bugs team, we believe this is mainly because the OWASP Benchmark developers made no effort to configure the tool parameters to achieve the best possible results by that tool. On the contrary, we did configure the parameters of Find-Security-Bugs to achieve a fair performance so that it becomes a competitive baseline in the experiments.

As it is visible in the results of Table 2, the Random-Forest model performs fairly better than Find-Security-Bugs in all vulnerability categories. Yet, in comparison to the proposed approach, the results clearly show a meaningful advantage in favor of the GNN models. The GCN model outperforms both baselines by a large margin in almost all categories. The GAT model also outperforms the baseline models by a meaningful margin, which again emphasizes the effectiveness of the proposed approach for the task at hand.

The advantage of proposed GNN models is less meaningful in two categories: CWE-90 and CWE-643; which can be explained because of the small number of test-cases in these two categories (see Table 1), resulting in poor learning quality. The few samples have an even more adverse effect on the performance of the GAT model, which results in lower performance compared to baseline models. Based on this observation, we will exclude these categories from later experiments.

Comparing the results of GCN and GAT, we can see that GCN achieves considerably better results in this setup. While the attention mechanism in the GAT model provides more learning capacity, this capability is more suitable for other setups (This is more thoroughly investigated in Section 5.5).

Insight **5.1.** *Experimental results conclude that the proposed approach based on graph neural network models outperforms baseline methods when trained with enough samples; hence can be utilized effectively for vulnerability analysis using intermediate graph representations of programs.*

5.2. Effect of various program graphs

RQ-2: **How various graph-representations of programs affect the performance of the proposed approach for the task of software vulnerability analysis?**

In the second research question, we aim to understand the effect of different graph representations of programs on the performance of GNN models for the task at hand. To this end, we will run our experiments with four different graph representations (AST, CFG, DDG, and CDG) for all vulnerability categories and compare the resulting F1-scores as the evaluation metric. As in Section 5.1, we use the same TF-IDF with SVD method with tensor size set to 256 for embedding code attributes in graph vertices. Experimental results are presented in Table 3.

The results for all categories and models show that the best results are achieved using data dependence graphs (DDG). Results for control flow graphs (CFG) and DDG are very close, and CFG results are just slightly below DDG results. This observation was expected since our customized DDG is a CFG augmented with data-dependence edges. The slightly improved results of DDG compared to CFG can be explained with the additional semantic information available in our customized DDG.

Results for abstract syntax trees (AST) and control dependence graphs (CDG) have a meaningful margin with CFG and DDG. This observation was also expected and can be explained with the much less semantic information available in the AST structure. For CDG, we conclude that control-dependence information is not sufficient for effective vulnerability analysis; which is also confirmed in previous research [48].

Table 3
F1-score results for RQ-2 experiments.

Model	Graph	CWE-22	CWE-78	CWE-79	CWE-89	CWE-501
GCN	AST	.6419	.6474	.6929	.6175	.6483
	CFG	.9123	.9259	.8899	.9476	.9044
	DDG	.9262	.9375	.9071	.9548	.9310
	CDG	.5771	.6401	.7214	.6454	.8163
GAT	AST	.6339	.5986	.6695	.7072	.7918
	CFG	.7788	.8137	.8036	.8861	.8711
	DDG	.8156	.8554	.8116	.8865	.8713
	CDG	.6052	.6207	.6497	.7050	.7626

AST: Abstract Syntax Tree.
DDG: Data Dependence subgraph of PDG.
CFG: Control Flow Graph.
CDG: Control Dependence subgraph of PDG.

Insight **5.2.** *Experimental results conclude that the customized data dependence graphs (DDG) and control flow graphs (CFG) achieve the best performance for vulnerability analysis using graph neural network models, with only minor differences. Abstract syntax trees (AST) and control dependence graphs (CDG) achieve the least performance, with major differences compared to CFG and DDG.*

5.3. Effect of code attributes embedding method

RQ-3: **How various code embedding methods for vertex attributes affect the performance of the proposed approach for the task of software vulnerability analysis?**

In the third research question, we aim to understand the effect of different code embedding techniques for vertex attributes of graphs on the performance of the proposed approach for the task at hand. To this end, we will run our experiments on graphs for every vulnerability category using two different code embedding techniques and compare the resulting F1-scores as the evaluation metric. The embedding techniques selected for this purpose are the TF-IDF with SVD method that we used in previous experiments and the Doc2Vec distributed embedding technique. The tensor size is also varied to examine its effect. Based on the experimental results of Section 5.2, we use the customized CFG graphs for the experiments in this section. Experimental results are presented in Table 4.

The results for all vulnerability categories show that using TF-IDF with the SVD technique achieves significantly better results for both GCN and GAT models compared to using the Doc2Vec distributed embedding technique. This observation was also true for the baseline Random-Forest (RFT) ensemble-learning method. This may seem surprising because of the opposite situation in the domain of natural language processing (NLP), where distributed embedding techniques such as Doc2Vec outperform classic latent semantic analysis techniques. One important fact to consider is that programming languages are different from natural languages, and the nature of program analysis is also different from natural language corpora analysis [3]. Yet, as we shall see in Section 5.5, the capabilities of distributed embedding techniques will be evident in the challenging cross-project setup.

The results also show that the tensor size of vertex attributes has a significant effect on the models' performance. For the baseline Random-Forest (RFT) model, a smaller tensor size achieves better results when TF-IDF with SVD is used. When using the distributed Doc2Vec technique, the results are not conclusive of the effect of tensor size. The pattern varies in different vulnerability categories, which means the type of vulnerability may demand a different tensor size. However, experimental results for GNN models show that a 64 tensor size generally performs worse compared to 128 or 256 tensor sizes. Best results for the GCN model are achieved with 128, while the GAT model performs better with 256 tensor-size.

Insight **5.3.** *Experimental results conclude that the code embedding technique utilized for vertex attributes of the graphs has a significant effect on the performance of software vulnerability analysis using graph neural network models. In particular, experimental results in this study show that TF-IDF with SVD outperforms the Doc2Vec method for the within-project setup.*

Moreover, the tensor size of vertex attributes also has a significant effect on the performance of graph neural network models. Too small tensor sizes can hinder the model's performance. In particular, experimental results in this study show that a tensor size of 128 or 256 is more effective compared to 64.

5.4. Effect of code normalization

RQ-4: **How does code normalization affect the performance of the proposed approach for the task of software vulnerability analysis?**

In the fourth research question, we aim to understand the effect of code normalization for the vertex attributes of graphs on the performance of GNN models for the task at hand. By code normalization, we mean to replace all variable identifiers

Table 4
F1-score results for RQ-3 experiments.

Model	Embedding	CWE-22	CWE-78	CWE-79	CWE-89	CWE-501
RFT	TF/IDF-64	.8433	.8044	.8132	.8196	.7455
	Doc2Vec-64	.5710	.5378	.6462	.5258	.6978
	TF/IDF-128	.8584	.7884	.7780	.8117	.7138
	Doc2Vec-128	.6043	.5022	.6615	.6052	.6271
	TF/IDF-256	.8024	.7606	.7495	.7681	.7305
	Doc2Vec-256	.5706	.5379	.6549	.6210	.6511
GCN	TF/IDF-64	.8034	.7831	.8002	.8852	.8947
	Doc2Vec-64	.6716	.6756	.6984	.7673	.8285
	TF/IDF-128	.9571	.9441	.9126	.9551	.9403
	Doc2Vec-128	.6990	.6645	.6749	.8360	.8010
	TF/IDF-256	.9123	.9259	.8899	.9476	.9044
	Doc2Vec-256	.6693	.6064	.7357	.7798	.8031
GAT	TF/IDF-64	.6047	.6363	.7207	.6786	.8616
	Doc2Vec-64	.6099	.6163	.6900	.7085	.8206
	TF/IDF-128	.7031	.7245	.8105	.8213	.8278
	Doc2Vec-128	.6070	.6947	.6426	.7207	.7886
	TF/IDF-256	.7788	.8137	.8036	.8861	.8711
	Doc2Vec-256	.6493	.6381	.6558	.6939	.8314

with a common placeholder. This is to see the effect of the variable names on the analysis performance. While program variable names convey important information to the human reader (contain natural language information), they have no effect on the program semantics. A vulnerable program whose variable identifiers are renamed to meaningless identifiers is still vulnerable.

To this end, we will run our experiments on graphs for every vulnerability category using normalized code attributes and compare resulting F1-scores as the evaluation metric. For variable normalization, we used "`$VAR`" as a common placeholder to rename all variable identifiers in the sample programs. We implemented the variable normalization process in PROGEX for AST graphs, and hence we perform our experiments only using AST graphs. Based on experimental results of Section 5.3, we use the TF-IDF with SVD embedding technique. Experimental results are presented in Table 5.

Results are clear; normalization has a negative effect on both GNN models, as well as the baseline model performance in all cases. What this means is that the models are using the natural language information that is conveyed via variable identifiers. This is consistent with previous research by Allamanis et al. [3], which survey previous studies that exploit the naturalness in computer programs.

Insight **5.4.** *Experimental results conclude that variable identifier normalization has a negative effect on the performance of both the baseline machine learning method, as well as the proposed approach using graph neural network models for software vulnerability analysis.*

5.5. Cross-project generalization capability of GNN

RQ-5: **How effective are GNN models for the task of cross-project software vulnerability analysis?**

In the fifth research question, we aim to understand the generalization capability of the trained GNN models for the challenging task of cross-project software vulnerability analysis. In other words, "*how prone the proposed approach is to overfitting?*". Can the models operate with acceptable performance when trained on a fairly different set of samples?.

To this end, we shall first perform model training and validation using the samples of OWASP Benchmark v1.1, and then perform testing on the samples of v1.2 of the Benchmark test suite (see Table 1). It should be noted that while the differences between samples of v1.1 and v1.2 are not very drastic, they are fairly different for the purpose of this experiment. In order to gain further insight, we conducted the experiments using both TF-IDF/SVD and Doc2Vec embeddings of size 256, on the customized CFG graphs. Also note that we have re-included CWE-90 and CWE-643 in these experiments, because of the adequate number of samples in the training set. As before, model validation is done using 5-fold cross-validation on the training dataset (v1.1 in this setup). Experimental results are presented in Table 6.

First, we analyze the validation results for each method with TF/IDF embedding. For the base-line Random-Forest method (RFT) we see some slight improvements compared to RQ-1 results of Table 2. This can be explained due to the fact that v1.1 has many more samples compared to v1.2 in all categories. The validation results for GCN and GAT (with TF/IDF) closely

Table 5
F1-score results for RQ-4 experiments.

Model	Embedding	Programs	CWE-22	CWE-78	CWE-79	CWE-89	CWE-501
RFT	TF/IDF-128	Original	.7802	.7968	.7780	.7799	.6905
		Normalized	.6797	.5815	.7451	.7165	.6668
	TF/IDF-256	Original	.7389	.7449	.7187	.7582	.6905
		Normalized	.6382	.5378	.7011	.6848	.6748
GCN	TF/IDF-128	Original	.6112	.6342	.6940	.6805	.8455
		Normalized	.5995	.6318	.6570	.6705	.7833
	TF/IDF-256	Original	.6345	.6551	.6732	.6718	.7100
		Normalized	.5697	.6442	.6641	.6672	.6650
GAT	TF/IDF-128	Original	.6229	.6715	.7104	.6962	.8070
		Normalized	.5675	.5808	.6632	.6690	.8047
	TF/IDF-256	Original	.6486	.6496	.7137	.7024	.7709
		Normalized	.6334	.5974	.6837	.6903	.7202

resemble results from RQ-1 (see Table 2). While minor improvements are visible in most of the categories, the results for CWE-90 and CWE-643 have major improvements, which is clearly because of the much larger number of training samples, yielding more effective learning. Validation results using Doc2Vec embedding has a similar outcome, and we see somewhat significant improvements compared to RQ-3 results in Table 4, which again can be explained due to the much larger number of training samples of v1.1 compared to v1.2.

The more interesting results that answer RQ-5 are the results for "Testing". A common observation in all methods and categories is a significant drop of F1-scores for testing compared to validation. Of course, this was expected, given the fact that the training dataset is fairly different from the testing dataset. Key observations from the cross-project testing results in Table 6 are as follows:

1. The GCN model achieves excellent results for validation (within-project setup); yet, it presents much poor testing results in the cross-project setup (in some cases, even worse than the base-line method). What this means is that GCN is much prone to over-fitting the training dataset, and this should be considered when designing solutions. More specifically, in terms of *precision* and *recall* values (Eq. 8), GCN achieved moderate *precision*, but poor *recall*; resulting poor F1-scores for the cross-project setup. These results are consistent with the results and insights of RQ-1 and RQ-3.
2. The GAT model shows significantly better cross-project testing results compared to GCN (as well as the base-line). This difference is actually inline with the underlying theories of GAT. As explained in Section 3.5, the attention mechanism of GAT provides more capable inductive bias. In the case of our experiments, we observe overall excellent within-project performance (see Section 5.1) but consistently lower compared to GCN in all experiments; yet, providing more generalization capability by consistently outperforming GCN in all categories of the cross-project setup. Again, in terms of *precision* and *recall* values (Eq. 8), GAT achieved good *precision*, and even better *recall*; resulting overall pleasing F1-scores for the cross-project setup. These results are consistent with the results and insight of RQ-1 and RQ-3.
3. Doc2Vec embedding also shows significantly better cross-project performance compared to TF/IDF-SVD embedding. This is consistent with the theories underlying the two techniques, as TF/IDF embedding is much more reliant on the exact terms used in documents (program source-code) and hence performs much better in the within-project setups of all our experiments; whereas Doc2Vec is a distributed embedding technique and is much more robust when facing variations in the programs source-code. These results are also consistent with the results and insight of RQ-3.

Insight **5.5.** *Experimental results conclude that the GAT model has much better generalization capability compared to both GCN and the chosen base-line method and shows prime potential as an acceptable solution for the cross-project setup. On the other hand, GCN seems to be much prone to over-fitting and seems to be a more suitable fit for within-project setups.*

Moreover, the TF/IDF embedding technique presents excellent results in a within-project setup, whereas the Doc2Vec distributed embedding technique is a more robust choice for cross-project situations facing more variations in project source-code.

5.6. Threats to validity

As with any research study, we shall discuss threats to the validity of our conclusions. An important note here is that while validation of hypotheses using empirical observations is an essential step in the scientific method [17], successful empirical results do not prove the hypotheses to be absolute facts.

Table 6
F1-score results for RQ-5 experiments (Validation on v1.1 and Testing on v1.2).

		Method	CWE-22	CWE-78	CWE-79	CWE-89	CWE-90	CWE-501	CWE-643
Validation	RFT	TF/IDF-256	.7901	.7718	.6988	.7756	.8084	.8069	.7586
		Doc2Vec-256	.7027	.6905	.6928	.7053	.7188	.7943	.6255
	GCN	TF/IDF-256	.9413	.9403	.8587	.9449	.9363	.9281	.9154
		Doc2Vec-256	.8248	.8251	.8234	.8273	.8416	.8414	.8128
	GAT	TF/IDF-256	.8698	.8577	.7925	.8765	.8903	.8748	.8551
		Doc2Vec-256	.7742	.7923	.7308	.7838	.8135	.8386	.7828
Testing	RFT	TF/IDF-256	.5847	.5686	.4254	.6795	.5926	.7021	.4889
		Doc2Vec-256	.6133	.6649	.6225	.6807	.6353	.7196	.6013
	GCN	TF/IDF-256	.5306	.5533	.5149	.5279	.5262	.6335	.5517
		Doc2Vec-256	.6475	.6748	.6044	.5853	.6279	.7588	.6027
	GAT	TF/IDF-256	.6617	.6558	.5580	.6868	.5952	.7875	.6049
		Doc2Vec-256	.6633	.6703	.6279	.7019	.6428	.7987	.6222

An important limitation to the generality of our conclusions is the utilization of a suite of synthetic programs in order to perform our experiments. The OWASP Benchmark is a high-quality suite of sample programs developed by domain experts. The latest version of the suite is fully executable and the vulnerabilities are exploitable. In order to be used as a benchmark suite, it is critical to keep the experimental setup in control and avoid unnecessary complexities that might affect experimental results in unknown ways. The OWASP Benchmark suite is designed with this point in mind; hence the samples are rather small compared to real-world applications, yet at the same time it is enough realistic to be a fully executable and exploitable web application. This makes the OWASP Benchmark a suitable option for research efforts.

Nevertheless, the need for a fully realistic set of vulnerable and secure programs is clear and present. The realization of such a data-set requires substantial effort for gathering real-world samples and associated meta-data. Unfortunately, we did not have enough resources to accomplish this task in the course of our study. We acknowledge that experiments on a real-world suite of programs greatly improve the generality of our conclusions; yet, the presented results are valuable contributions to the research community and can shed light for future studies to further continue this path.

6. Conclusion

In this study, we put forth an original hypothesis that utilizing rich graph representations and modern graph neural networks is an effective approach for the task of software vulnerability analysis. To verify our hypothesis, a novel neural vulnerability analysis approach was proposed, based on modern graph neural networks. Rich graph representations of programs were extracted using a tool developed in the course of this study and utilized for training two graph neural network models. Experimental results show that the proposed approach is clearly effective by outperforming some well-known baseline methods by a large margin. This is achieved using only a few hundred training samples, compared to modern deep neural networks that require several thousands of training samples to be effective. Experimental results also showed that the proposed approach performs effectively for cross-project vulnerability analysis. Moreover, the PROGEX tool is publicly published with a liberal license as an additional contribution to the research community.

Future work for improvement remains to be pursued in this path. A dataset of real-world vulnerability cases and their patched variant, accompanied by enough metadata is a valuable contribution to the research community to help evaluate the applicability of proposed approaches in the software industry. Experimenting with other types of graph neural networks, different architectures, and using different graph representations of programs is also another path for exploration. Another interesting open problem for future research is to develop explainable or interpretable graph neural network models.

CRediT authorship contribution statement

Seyed Mohammad Ghaffarian: Conceptualization, Methodology, Software, Data curation, Writing - original draft. **Hamid Reza Shahriari:** Conceptualization, Validation, Writing - review & editing, Supervision, Project administration.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] C. Aggarwal, Data preparation, in: Data Mining: The Textbook, Chapter 17, Springer, 2015, pp. 557–587..
[2] A. Aho, M. Lam, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, second ed., Addison-Wesley, 2006.
[3] M. Allamanis, E. Barr, P. Devanbu, C. Sutton, A survey of machine learning for big code and naturalness, ACM Comput. Surveys (CSUR) 51 (4) (2018) 81:1–81:37..
[4] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, in: Proceedings of the 6th International Conference on Learning Representations (ICLR), 2018.
[5] N. Ayewah, D. Hovemeyer, D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, IEEE Software 25 (5) (2008) 22–29.
[6] P. Battaglia, J. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al., Relational Inductive Biases, Deep Learning, and Graph Networks. Computing Research Repository (CoRR); arXiv:1806.01261, 2018..
[7] M. Brockschmidt, M. Allamanis, A. Gaunt, O. Polozov, Generative code modeling with graphs, in: Proceedings of the 7th International Conference on Learning Representations (ICLR), 2019.
[8] M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, P. Vandergheynst, Geometric deep learning: going beyond Euclidean data, IEEE Signal Process. Mag. 34 (4) (2017) 18–42.
[9] R.Y. Chang, A. Podgurski, J. Yang, Discovering neglected conditions in software by mining dependence graphs, IEEE Trans. Software Eng. 34 (5) (2008) 579–596.
[10] H. Cheng, D. Lo, Y. Zhou, X. Wang, X. Yan, Identifying bug signatures using discriminative graph mining, in: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA), 2009, pp. 141–152.
[11] N. Dehmamy, A.-L. Barabasi, R. Yu, Understanding the representation power of graph neural networks in learning graph topology, in: Proceedings of the 33rd Conference on Neural Information Processing Systems (NIPS), 2019, pp. 15387–15397.
[12] F. Eichinger, K. Bohm. Software-bug localization with graph mining, in: Managing and Mining Graph Data, 2010, pp. 515–546..
[13] P. Foggia, G. Percannella, M. Vento, Graph matching and learning in pattern recognition in the last 10 years, Int. J. Pattern Recogn. Artif. Intell. 28 (01) (2014).
[14] S.M. Ghaffarian, H.R. Shahriari, Software vulnerability analysis and discovery using machine learning and data mining techniques: a survey, ACM Comput. Surveys (CSUR) 50 (4) (2017) 56:1–56:36.
[15] A. Habib, M. Pradel, How many of all bugs do we find? A study of static bug detectors, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), 2018, pp. 317–328.
[16] A. Habib, M. Pradel, Neural Bug Finding: A Study of Opportunities and Challenges. Computing Research Repository (CoRR); arXiv:1906.00307, 2019..
[17] C. Herley, P. van Oorschot, SoK: science, security and the elusive goal of security as a scientific pursuit, in: Proceedings of the 38th IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, 2017, pp. 99–120..
[18] S. Horwitz, T. Reps, The use of program dependence graphs in software engineering, in: Proceedings of the 14th International Conference on Software Engineering ICSE, ACM Press, 1992, pp. 392–411..
[19] I. Jana, A. Oprea, AppMine: behavioral analytics for web application vulnerability detection, in: Proceedings of the ACM Conference on Cloud Computing Security Workshop (CCSW), 2019, pp. 69–80.
[20] T. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, in: Proceedings of the 5th International Conference on Learning Representations (ICLR), 2017.
[21] A. Krizhevsky, I. Sutskever, G. Hinton, ImageNet classification with deep convolutional neural networks, in: Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS), Curran Associates Inc., 2012, pp. 1097–1105..
[22] W. Landi, Undecidability of static analysis, ACM Lett. Program. Lang. Syst. (LOPLAS) 1 (4) (1992) 323–337.
[23] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: Proceedings of the 31st International Conference on Machine Learning (ICML), 2014, pp. 1188–1196.
[24] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.
[25] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, in: Proceedings of the 4th International Conference on Learning Representations (ICLR), 2016.
[26] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, H. Jin, A comparative study of deep learning-based vulnerability detection system, IEEE Access 7 (2019) 103184–103197.
[27] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, J. Wang, SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. Computing Research Repository (CoRR); arXiv:1807.06756, 2019..
[28] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: a deep learning-based system for vulnerability detection, in: Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), 2018.
[29] H. Liang, L. Sun, M. Wang, Y. Yang, Deep learning with customized abstract syntax tree for bug localization, IEEE Access 7 (2019) 116309–116320.
[30] L. Livi, A. Rizzi, The graph matching problem, Pattern Anal. Appl. 16 (3) (2013) 253–283.
[31] C. Manning, P. Raghavan, H. Schutze, Introduction to Information Retrieval, Cambridge University Press, 2008.
[32] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS), 2013, pp. 3111–3119.
[33] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for programming language processing, in: Proceedings of the 13th AAAI Conference on Artificial Intelligence, 2016, pp. 1287–1293.
[34] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, New York, 1999.
[35] S. Parsa, S. Arabi-Naree, N. Ebrahimi-Koopaei, Software fault localization via mining execution graphs, in: Proceedings of the International Conference on Computational Science and its Applications (ICCSA), 2011, pp. 610–623.
[36] M. Pradel, K. Sen, DeepBugs: a learning approach to name-based bug detection, in: Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2018, pp. 147:1–147:25..
[37] T. Reps, Program analysis via graph reachability, Inf. Software Technol. 40 (11) (1998) 701–726.
[38] T. Reps, S. Horwitz, S. Sagiv, Precise interprocedural dataflow analysis via graph reachability. in: 22nd ACM Symposium on Principles of Programming Languages POPL, ACM Press, 1995, pp. 49–61..
[39] F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Trans. Neural Networks 20 (1) (2009) 61–80.
[40] H. Shahriar, M. Zulkernine, Mitigating program security vulnerabilities: approaches and challenges, ACM Comput. Surveys (CSUR) 44 (3) (2012) 11.
[41] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G.V.D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al, Mastering the game of go with deep neural networks and tree search, Nature 529 (7587) (2016) 484.
[42] I. Sutskever, O. Vinyals, Q. Le, Sequence to sequence learning with neural networks, in: Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS), MIT Press, 2014, pp. 3104–3112..
[43] Y. Taigman, M. Yang, M. Ranzato, L. Wolf, DeepFace: closing the gap to human-level performance in face verification, in: Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE Computer Society, 2014, pp. 1701–1708..
[44] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, in: Proceedings of the 6th International Conference on Learning Representations (ICLR), 2018.
[45] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P. Yu, A Comprehensive Survey on Graph Neural Networks. Computing Research Repository (CoRR); arXiv:1901.00596, 2019..

[46] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks?, in: Proceedings of the 7th International Conference on Learning Representations (ICLR), 2019

[47] E. Yahav, From programs to interpretable deep models and back, in: Proceedings of the 30th International Conference on Computer Aided Verification (CAV), 2018, pp. 27–37.

[48] F. Yamaguchi, Pattern-Based Vulnerability Discovery. PhD thesis, Georg-August-University of Gottingen, Germany, 2015..

[49] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, M. Sun, Graph Neural Networks: A Review of Methods and Applications. Computing Research Repository (CoRR); arXiv:1812.08434, 2018..

[50] D. Zou, S. Wang, S. Xu, Z. Li, H. Jin, μ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. Computing Research Repository (CoRR); arXiv:2001.02334, 2020..