# Compact Abstract Graphs for Detecting Code Vulnerability with GNN Models

Yu Luo
University of Missouri-Kansas City
Kansas City, MO, USA
ylzqn@mail.umkc.edu

Weifeng Xu
The University of Baltimore
Baltimore, MD, USA
wxu@ubalt.edu

Dianxiang Xu
University of Missouri-Kansas City
Kansas City, MO, USA
dxu@umkc.edu

## ABSTRACT

Source code representation is critical to the machine-learning-based approach to detecting code vulnerability. This paper proposes Compact Abstract Graphs (CAGs) of source code in different programming languages for predicting a broad range of code vulnerabilities with Graph Neural Network (GNN) models. CAGs make the source code representation aligned with the task of vulnerability classification and reduce the graph size to accelerate model training with minimum impact on the prediction performance. We have applied CAGs to six GNN models and large Java/C datasets with 114 vulnerability types in Java programs and 106 vulnerability types in C programs. The experiment results show that the GNN models have performed well, with accuracy ranging from 94.7% to 96.3% on the Java dataset and from 91.6% to 93.2% on the C dataset. The resultant GNN models have achieved promising performance when applied to more than 2,500 vulnerabilities collected from real-world software projects. The results also show that using CAGs for GNN models is significantly better than ASTs, CFGs (Control Flow Graphs), and PDGs (Program Dependence Graphs). A comparative study has demonstrated that the CAG-based GNN models can outperform the existing methods for machine learning-based vulnerability detection.

## CCS CONCEPTS

• **Security and privacy → Software and application security**.

## KEYWORDS

Software vulnerability, machine learning, graph neural networks, static code analysis

**ACM Reference Format:**
Yu Luo, Weifeng Xu, and Dianxiang Xu. 2022. Compact Abstract Graphs for Detecting Code Vulnerability with GNN Models. In *Annual Computer Security Applications Conference (ACSAC), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3564625.3564655


*ACSAC'22, December 5–9, 2022, Austin, TX*

ACM ISBN 978-1-4503-9759-9/22/12…$15.00
https://doi.org/10.1145/3564625.3564655

## 1 INTRODUCTION

Software vulnerability is a major source of cybersecurity risks. For example, 419 of the 924 Common Weakness Enumerations (CWE) fall into the category of software vulnerability. While academia and industry have devoted significant effort to cybersecurity, new vulnerabilities are continually identified in various products. Many are publicly disclosed via the Common Vulnerabilities and Exposures (CVE) [8] and National Vulnerability Database (NVD) [33]. In 2021, the number of vulnerability reports in NVD had increased to 20,138, up 9.7% from 2020.

Detecting software vulnerability is a challenging task. Traditional approaches include static analysis and dynamic testing of binary [50] [16] [52] [5] [17] and source code [6] [31] [15] [21] [4]. They usually target specific types of code vulnerability. For instance, code clone-based methods are limited to the vulnerabilities caused by code cloning. Pattern-based methods rely on certain vulnerability rules pre-defined by human experts. As a promising technique for static code analysis, machine learning has gained increasing attention [41] [53] [43] [32] [19] [34]. The existing work falls into three categories: (a) treating source code as text to exploit NLP (Natural Language Processing) models (e.g., N-gram and bag-of-words), (b) extracting different paths from the source code's abstract syntax trees (ASTs) for learning from sequence samples. (c) representing source code as graphs to apply Graph Neural Network (GNN) models.

This paper aims to address the practical issue of source code representation for exploiting the state-of-the-art GNN models to detect a broad range of code vulnerabilities. We propose Compact Abstract Graphs (CAGs) of source code in different programming languages (e.g., Java and C) for efficient vulnerability prediction with various GNN models. We construct the CAG of a routine (Java method or C function) from its AST (Abstract Syntax Tree) in two stages: (a) converting the AST into an Abstract Graph (AG), which keeps all AST nodes with reversed edges and connects each token node to the AST root and the next token, (b) reducing the AG by merging the longest sequences of single-entry property nodes and the aggregation structures. The first stage makes the source code representation aligned with the GNN-based vulnerability classification, which aims to map the code features (e.g., source code tokens and their relations) to pre-defined vulnerability labels (e.g., "positive" for the existence and "negative" for the absence). The second stage reduces the graph size to accelerate GNN model training with minimum impact on their prediction performance (e.g., with little sacrifice of accuracy and precision).

We have applied CAGs to six GNN models: Graph Convolutional Networks (GCNs) [20], Graph Attention Networks (GATs) [46], Unified Message Passing Model (UniMP) [44], GNNs with

auto-regressive moving average filter (ARMAConv) [2], Residual Gated Graph ConvNets (RGGCNs) [3], and Feature-Steered Graph Convolutions (FeaStNet) [47]. They represent various strategies for graph modeling and learning. To evaluate the approach, our Java dataset includes 37,350 vulnerable methods (covering 114 vulnerability types) and 68,480 non-vulnerable methods and our C dataset has 58,459 vulnerable functions of 106 vulnerability types and 126,170 non-vulnerable functions. The samples are collected from NVD [33], Software Assurance Reference Dataset (SARD) [40] and other publications [30][48]. The experiment results show that all six models performed well, with accuracy ranging from 94.7% to 96.3% on the Java dataset and from 91.6% to 93.2% on the C dataset. The resultant GNN models have achieved promising performance when applied to more than 2,500 vulnerabilities collected from real-world software projects. The results also show that using CAGs for GNN models is significantly better than ASTs, CFGs (Control Flow Graghs), and PDGs (Program Dependence Graphs). Compared to AGs, CAGs reduce 53.7% nodes, 39.7% edges, and 54.9% file sizes. It saves 27.4% of the training time while preserving the prediction performance in terms of accuracy, precision, recall and F1 scores. Moreover, a comparative study has demonstrated that the CAG-based GNN models can outperform the existing methods for machine learning-based vulnerability detection. In brief, CAGs are an effective representation of source code for GNN-based vulnerability detection.

The remainder of this paper is organized as follows. Section 2 reviews related work; Section 3 describes the CAGs of source code; Section 4 introduces the framework for vulnerability detection with GNN models; Section 5 presents the experiment results; Section 6 applies the vulnerability prediction models to real-world programs. Section 7 concludes this paper.

## 2 RELATED WORK

This paper is related to the work on detecting code vulnerability via machine learning. It falls into three categories in terms of how the code is represented for machine learning: (a) code as text, (b) code as trees, and (c) code as graphs.

Text-based methods exploit NLP models by treating source code as text. Peng et al. [35] used the n-gram model to convert Java source code into vectors and optimized each vector through the Wilcoxon rank-sum [51]. Hovsepyan et al. [19] and Pang et al. [34] predicted software weakness using SVM on a bag-of-words (BOW) and n-grams representation of basic tokenization of Java source code. Scandariato et al. [41] presented a text-mining-based vulnerability prediction algorithm for Java applications. To build feature vectors, the original code is tokenized into monograms and combined with frequency. They used the Nave Bayes and Random Forest algorithms to forecast vulnerability. Lee et al. [23] developed an Instruction2vec model to capture vulnerable features from assembly codes and trained on CNNs to detect bugs in C programs. Yamaguchi et al. [53] applied NLP approaches to vulnerability assessment. This method captures API symbols, embeds API symbols in a vector space, and uses machine learning to predict API use trends. Luo et al. [30] applied natural language syntax on Java source code and detected integer overflows through BERT [10]. Russell et al. [38] utilized CNNs and RNNs for extracting features from embedded source representations parsed by a custom lexer. A random forest classifier is trained to identify whether the C/C++ source code includes five sorts of weakness. Le et al. [22] proposed the maximal divergence sequential autoencoder as a features extractor and built a full connected neural network to detect vulnerabilities on binary code. Choi et al. [7] and Sestili et al. [42] encoded the raw source code line by line to one-hot vectors that used as the input of a memory network. The model contains an attention mechanism to locate buffer overflows. These methods focus on the semantic information of the source code and ignore the overall structural features of programs.

Tree-based approaches extract features by traversing the ASTs and their variants. Dong et al. [11] utilized code sequences extracted from ASTs as semantic features and the frequency as token features to build a full connected neural network for detecting vulnerabilities in Android binary executables. Wang et al. [49] preserved three types of nodes, and converted them into code sequences. These sequences are mapped into high dimension vectors to train deep belief networks (DBNs) for detecting software weakness. POSTER [27] and Lin et al. [28] discovers vulnerabilities in function level by building a bidirectional LSTM network with ASTs-based sequences. Dam et al. [9] built a sequence to sequence LSTM network to learn the semantic and syntactic features from ASTs in Java methods. Shabtai et al. [43] applied principal component analysis to the ASTs of source code to identify vulnerable code. Similarly, Mokhov et al. [32] used numerous methods from WEKA [18] and the ASTs as characteristics to construct prediction models. SySeVR [25] divided programs into small pieces and generated multiple representations from ASTs to exhibit the syntax and semantics characteristics of vulnerabilities. These approaches are incapable of capturing intricate program structural properties (branches or parallel statements).

Graph-based techniques generally capture structural features from multiple graphs, such as control-flow graphs (CFGs), data-flow graphs (DFGs), data dependence graphs (DDGs), etc., and utilize GNNs, CNNs or RNNs to build vulnerability detection models. DEVIGN [55] generated a joint graph representation of code snippets by merging ASTs, CFGs and DFGs. It built a GNN-based model with three layers (a graph embedding layer, a gated graph recurrent layer, and a convolutional layer) to detect vulnerabilities in C programs. FUNDED [48] created a method-level graph representation by creating nine types of edges to ASTs. The node embeddings are calculated through a pre-trained word2vec network updated by a gate recurrent unit (GRU). It has achieved an average accuracy of 92%. VulDeePecker [26] seeks to identify buffer errors and resource management mistakes in C/C++ programs. It extracts API function calls and forward/backward program slices from CDGs and DDGs. A forward slice relates to statements influenced by the argument, whereas a backward slice corresponds to statements that can affect the argument. uVulDeePecker [56] is an extension of VulDeePecker that extracted data and control dependencies from system dependency graphs (SDGs). The code attention and localized information help the model deal with 40 types of vulnerabilities. Wang et al. [29] trained a GNN model on contract graphs to detect smart contract vulnerabilities, which obtained an average accuracy of 89%.

In addition to vulnerability detection, there are other source code representations for program analysis. CodeBERT [14] is a

bimodal pre-trained model by adding programming language to base BERT [10], which supports downstream NL-PL applications. SourcererCC [39] transformed programs into regularized token sequences and ordered by optimized inverted index for code clone detection. Based on program CFGs and PDGs, Allamanis et al. [1] applied Gated Graph Neural Networks to predict variable names and detect variable misuses, and DeepSim [54] encoded flows into a semantic matrix for measuring code functional similarity.

## 3 COMPACT ABSTRACT GRAPHS

The proposed compact abstract graphs of source code are a compact representation of abstract graphs built from the abstract syntax trees (ASTs) of source code. In the following, we first introduce the abstract graphs and discuss how to compress them by merging sequential and aggregation structures.

### 3.1 Abstract Graphs

The abstract graph (AG) of a routine (Java method or C function) is a triple $\langle N, E, s\rangle$, where $N$ is a set of nodes, $E$ is a set of directed edges, and $s \in N$ is the global sink (with no exiting edges) reached by all directed edges. It is built from the routine's AST $\langle N_p, N_t, \mathbb{E}, r\rangle$, where $N_p$ is a set of non-terminal property nodes, $N_t$ is a set of terminal nodes (i.e., source code tokens), $r$ is the root representing the overall routine, and $\mathbb{E}$ is a set of directed edges between the nodes in $N_p \cup N_t \cup \{r\}$. A terminal token node represents a token in the routine's source code, whereas a property node specifies a syntactic property of the token connected through edges. The property nodes depend on the syntax of the underlying programming language. An edge from a token node to a property node means that the token has the property (e.g., "static" is a keyword and "int" is a type). An edge from property node $A$ to property node $B$ indicates that $A$ is more specific or $A$ is part of $B$. In general, we say $B$ is more abstract than $A$. For example, the edge ("keyword", "Modifier") means "keyword" is a "Modifier". A property node with multiple incoming edges means that the property consists of multiple components.

The AG of a routine has the same set of nodes as its AST, i.e., $N = N_p \cup N_t \cup \{r\}$. We also refer to $N_p$ and $N_t$ in the AG as property nodes and token nodes, respectively. The edges are created from the AST $\langle N_p, N_t, r, \mathbb{E}\rangle$ by: (1) reversing all edges in $\mathbb{E}$, (2) adding an edge from each token to the next, and (3) adding an edge to connect each terminal token node to the root. Formally, $E = \{(a, b) : (b, a) \in \mathbb{E}\} \cup \{(x, y) : y \text{ is the token next to } x \text{ for each } x \in N_t\} \cup \{(x, r) : x \in N_t\}$. As a consequence, the AST's root $r$ becomes the global sink of the AG (named "MethodDeclaration"). It has no exiting edges and is reached by all edges in $E$.

Figure 1 shows a sample AG of Java method *sum*.

```
public static int sum(int[] numbers) {
    int total = 0;
    for (int number : numbers) {
        total += number;
    }
    return total;
}
```

The text in all the token nodes, such as "static", "public", and "int", comprises the given source code. Different from the AST, each token node in the AG has three exiting edges that lead to their immediate property node, next token node, and the global sink (the overall method declaration). For example, the immediate property of "static" is "keyword". An edge from property node $A$ to property node $B$ represents a higher level of syntactical abstraction. For example, the edge ("keyword", "Modifier") coming from the token node "static" means that the keyword "static" is a "Modifier", which further becomes part of the "modifiers" component of a method declaration. In general, the path from a token node to the global sink (e.g., <"static", "keyword", "Modifier", "modifiers", "MethodDeclaration">) represents a sequence of increasing abstractions that depicts how the source code token (e.g., "static") contributes to the overall method declaration. In the corresponding AST, however, each path goes from the root to a token node (i.e., from the most general to the most specific). In comparison, the aggregative paths in the AG are more aligned with the GNN-based classification task for determining whether the source code tokens contribute to the existence or absence of vulnerability in the entire method declaration. As demonstrated in Section 5, AGs have outperformed ASTs when they are applied to GNN models for vulnerability prediction.

The edges from each token node to the next corresponds to how the source code is written. Obviously, such ordering dependency is critical to effective code analysis. It must be represented explicitly in the graph representation for GNN-based vulnerability prediction.

As discussed above, the path from a token node to the global sink indicates how the token contributes to the overall method declaration. The contributions of source code tokens are more significant to the detection of vulnerability. For example, CWE-547 occurs because of using hard-coded constants instead of symbolic names for security-critical values, which increases the risk of mistakes during code maintenance. Therefore, the AG connects each token node directly to the overall declaration node (i.e., the sink).

### 3.2 Merging Single-Entry Node Sequences

We start reducing an abstract graph $\langle N, E, s\rangle$ by finding all longest sequences of single-entry property nodes and merging each sequence into one node. A longest sequence of single-entry property nodes is a list of property nodes $\langle n_1, n_2, ...n_k\rangle$ such that:

- Each $n_i \in N$ $(0 < i \leq k)$ is a property node with exactly one entry edge.
- $(n_i, n_{i+1})$ $(0 < i < k)$ is an edge in $E$.
- $\alpha$, the node connecting to the sequence's first node $n_1$, is either a token node or a property node with at least two entry edges.
- $\omega$, the node connected from the sequence's last node $n_k$ is either the sink node or a property node with at least two entry edges.

Figure 2 shows two typical sequence patterns, where $\alpha$ in (a) is a token node (with no entry edge) and $\alpha$ in (b) is a property node with multiple entry nodes. Consider the rightmost node sequence ⟨"identifier", "SimpleName", "NameExpr", "ReturnStmt"⟩ in Figure 1. "total" is the token node connected to the sequence's first node "identifier"; whereas "statements", the node connected from the sequence's last node, has three entry edges. The above sequence represents a sequence of abstractions of "token" (i.e., source code "return total"). We merge the sequence into a new node labeled

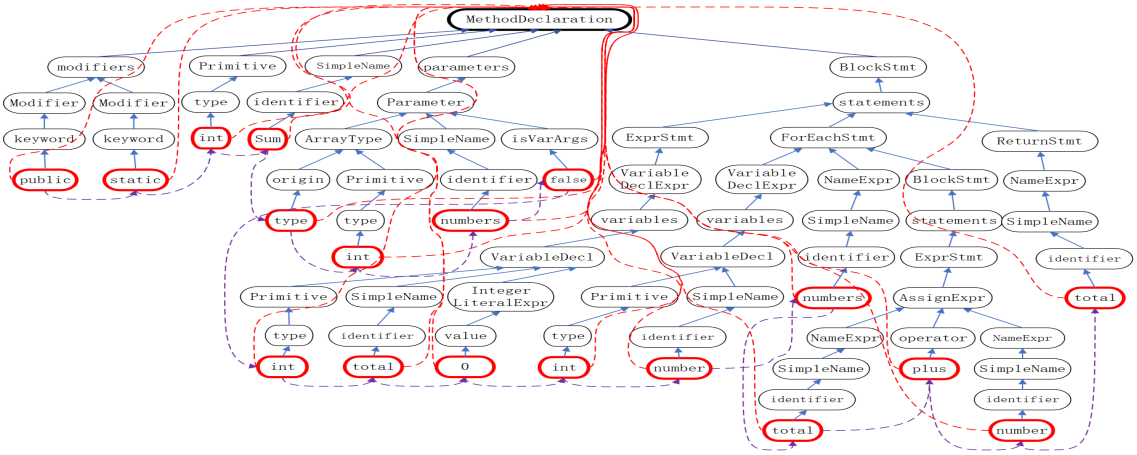



**Figure 1: The AG of a Java Method**

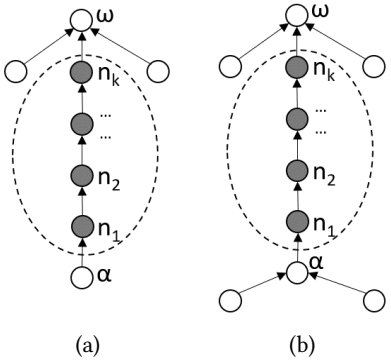



**Figure 2: Patterns of Single-Entry Node Sequences**

by the concatenation of all node labels (to preserve the abstraction information), replace edge ("total", "identifier") with ("total", new node), and edge ("ReturnStmt", "statements") with (new node, "statements").

Formally, given abstract graph $\langle N, E, s\rangle$ and a node sequence $\langle n_1, n_2, ...n_k\rangle$, merging the sequence results in a new abstract graph $\langle N', E', s\rangle$ such that $N' = N \setminus \{n_1, n_2, ...n_k\} \cup \{n\}$ and $E' = E \setminus \{(n_1, n_2), (n_2, n_3), ..., (n_{k-1}, n_k)\} \cup \{(\alpha, n), (n, \omega)\}$, where $n$ is the new merged node, $(\alpha, n_1) \in E$, $(n_k, \omega) \in E$. As discussed in Section 4, the encoding of a merged node into numeric data for GNN models is obtained from the encodings of all nodes.

The abstract graph in Figure 1 has 19 node sequences. Merging these sequences leads to the new graph in Figure 3.

## 3.3 Merging Aggregation Structures

We further reduce $\langle N', E', s\rangle$ by locating all compressible aggregation structures and merging each aggregation into one node. An aggregation structure $\langle \tau, n_1, n_2, ...n_k\rangle$ represents a child-parent relation, where $\tau$ is the "parent" and its children are $n_1, n_2, ...n_k$. The structure satisfies the following conditions:

- $\tau \in N' \setminus \{s\}$ is a property node with two or more entry edges.
- Each $n_i \in N'$ $(0 < i \leq k)$ is a property or sequence merged node connecting to $\tau$.
- $(n_i, \tau)$ $(0 < i \leq k)$ is an edge in $E'$.
- Each $A_i \in \langle A_1, A_2, ...A_k\rangle$ $(0 < i \leq k)$ is a list of property or token node connecting to $n_i$, $\{(\alpha, n_i) : \alpha \in A_i\} \subset E'$.
- $\omega$, the node connected from node $\tau$ is either the sink node or has one or more entry edges.

Figure 4 shows the aggregation patterns. Whether an aggregation structure can be compressed depends on the following conditions. (1) Each child node $n_i$ in the structure has exactly one entry edge. If any child node has two or more entry edges, we cannot merge the aggregation structure; Otherwise, it would lose structural information. (2) the structure represents the part-whole relation of a programming construct as detailed below. In Figure 4, (a) is compressible, but (b) is not compressible.

The aggregation structures fall into seven categories: expression, statement, declaration, argument, parameter, type, and modifier. Each category has various nodes as shown in Table 1.

The expression structure is composed of one or more constants, variables, functions, and operators. In an expression aggregation, the parent node is connected by all child nodes to show the whole expression. We can merge it into one node without losing critical information. For example, the "AssignExpr" node consists of two "NameExpr" nodes and one "operator" node. After merging, we use a node with above four elements to reflect the line of code "total += number;". Different from expressions, the child nodes in a statement aggregation (except return statement) presents the parallel relationship of each line of source code tokens in the same block. They are related and independent. Merging the nodes in a statement will lose information about their parallel connections. The return statement is a special statement that is similar to expression. All child nodes under a return statement form the entire return statement. Statement aggregations, except the return statement,cannot be merged.

In the declaration category, 'variabledeclarator' shows the process of declaring a variable. It can be merged. Most of the argument,

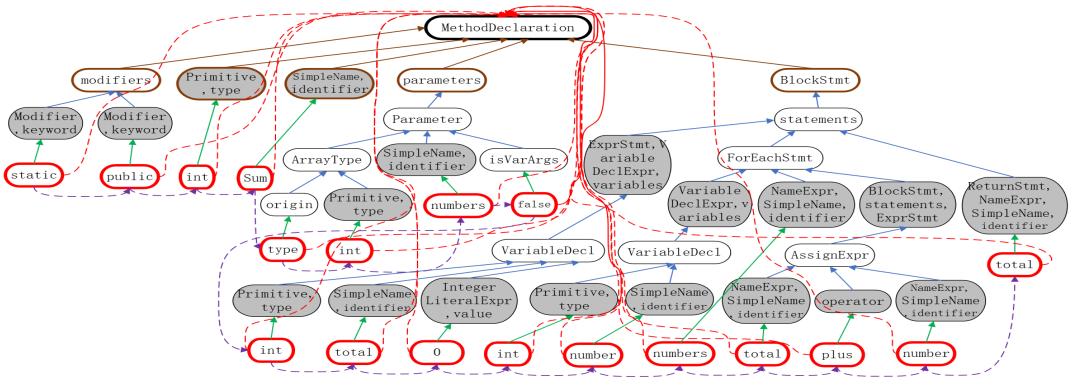



**Figure 3: The AG after Merging Single-Entry Node Sequences**

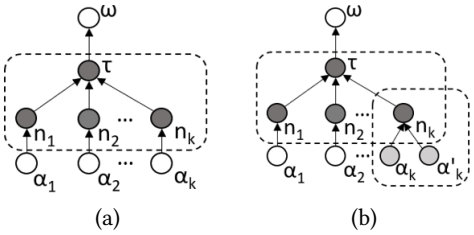



**Figure 4: Patterns of Aggression Structure**

**Table 1: Categories of Aggression Structures**

| Category | Label of the Parent Node |
|---|---|
| Expression | MethodCallExpr, FieldAccessExpr, BinaryExpr, ObjectCreationExpr, AssignExpr, ArrayCreation -Expr, CastExpr, ArrayAccessExpr, IntegerLiteral -Expr, UnaryExpr, InstanceOfExpr, SingleMember -AnnotationExpr, VariableDeclarationExpr, ConditionalExpr |
| Statement | statements, IfStmt, CatchClause, TryStmt, SwitchEntry, ForStmt, WhileStmt, catchClauses, entries, SwitchStmt, ExpressionStmt, ForEachStmt, LabeledStmt, DoStmt, AssertStmt, ThrowStmt, ReturnStmt, thrownExceptions, SynchronizedStmt |
| Declaration | VariableDeclarator, variables, values |
| Argument | arguments, typeArguments |
| Parameter | Parameter, parameters, typeParameters, TypeParameter |
| Type | ClassOrInterfaceType, ArrayType |
| Modifier | modifiers |

parameter and type categories comprise independent elements and thus cannot be merged.

The modifier category as a special aggregation of the routine signature can be merged. Because the source code tokens in a modifier are never merged, merging the rest of a modifier aggregation can still show the parallel relationship between the source code tokens.

**Algorithm 1** Compression of Aggression Structures

**Input:** $\langle N', E', s\rangle$, a set of mergeable aggregation types $L$, a function of merging node labels in an aggregation structure $M_a()$
**Output:** $CAG\langle N_c, E_c, s\rangle$

1: $N_p = \Phi$
2: **for** each node $n \in N'$ **do**
3: **if** $|\{(x, n) : x \in N'\} \cap E'| \geq 2$ **then**
4: $N_p = N_p \cup \{n\}$ ▷ $N_p$ is a list of parent nodes
5: **end if**
6: **end for**
7: **for** $\tau \in N_p$ **do**
8: **if** $\tau \in L$ **then**
9: $\{n_1, n_2, ..., n_k\} = \{n : \forall(n, \tau) \in E'\}$
10: **if** $\{n_1, n_2, ..., n_k\} \cap N_p = \Phi$ **then**
11: $m = M_a(\{n_1, n_2, ..., n_k\} \cup \{\tau\})$ ▷ $m$ is the new merged node
12: $E_c = E' \cup \{(m, \omega) : \forall(\tau, \omega) \in E'\} \cup \{(\alpha, m) : \forall(\alpha, n_i) \in E'$ for each $n_i \in \{n_1, n_2, ..., n_k\}\} \setminus \{(n_i, \tau) : n_i \in \{n_1, n_2, ..., n_k\}\}$
13: $N_c = N' \cup \{m\} \setminus (\{n_1, n_2, ..., n_k\} \cup \tau)$
14: **end if**
15: **end if**
16: **end for**
17: Return $CAG\langle N_c, E_c, s\rangle$

Algorithm 1 transforms the reduced AG $\langle N', E', s\rangle$ to a CAG by merging all compressible aggregation structures.

**Lines 1–6** extract the parent nodes $N_p$ of all aggregation structures. In $N'$, a node with two or more entry edges could be the parent node $\tau$ of an aggregation structure. The global sink is not a compressible aggregation.

**Lines 7–17** generate $CAG\langle N_c, E_c, s\rangle$ by merging all compressible aggregation structures. First, lines 7–10 determine whether an aggregation structure $\{\tau, n_1, n_2, ..., n_k\}$ is compressible. If so, line 11 creates the new merged node $m$ by combining all node labels in $\{\tau, n_1, n_2, ..., n_k\}$ through the function $M_a()$. In line 12, we create new edges: (1) $\{(m, \omega) : \forall(\tau, \omega) \in E'\}$, connecting the new node $m$ to all nodes connected by $\tau$, (2) all nodes point to each

node in $\{n_1, n_2, ..., n_k\}$ will connect to the new node $m$. Finally, we remove all nodes in $\{\tau, n_1, n_2, ..., n_k\}$ and edges in $\{(n, \tau) : n \in \{n_1, n_2, ..., n_k\}\} \cup \{(\tau, \omega) \in E'\}$.

The CAG of the above sample code is shown in Figure 5. It has eight aggregation structures. Four of them are compressed: one "modifiers", two "VariableDecl" , and one "AssignExpr" (i.e., the nodes in blue). The aggregation structures of "ArrayType", "Parameter", "statements", and "ForEachStmt" are not compressible.

# 4 VULNERABILITY DETECTION WITH GNN MODELS

## 4.1 The Framework

Figure 6 shows the general framework of our approach. It involves three phases: (1) preprocessing: creating the CAG of each routine in source code repository, (2) embedding: aggregating and embedding the nodes of each CAG to obtain a numeric code matrix for graph learning, and (3) modeling: training a GNN model to construct a predictive model. The model can then be used to determine whether an unseen program has vulnerabilities.

**Preprocessing.** To generate the CAG of each routine in the source code repository, we first parse the routine into an AST, and build the AG. Then we reduce it to the CAG by merging the sequences of single-entry property node and compressible aggregation structures as discussed in the previous section.

**Embedding.** Each graph (e.g., CAG or AG) is converted into a numerical representation before it is fed into the GNN layer. It consists of nodes, edges, and labels. We use MPNet [45] to embed each node token to a 768 fixed-length numeric vector. The idea is to construct a high dimensional space so that tokens with similar semantic features are mapped to the same area of space. For example, "string" and "int" are both variable's types, so their embedding vectors are close in the space. In the AGs, the text of each node originates from the ASTs, so it can be encoded into a vector through the MPNet directly. In the CAGs, there are three forms of nodes. The original node without merging directly embeds into a numeric vector like the AG nodes. For a node that represents a merged node sequence, its embedding vector $V_s$ is given in equation (1), where $n$ is the number of nodes in the sequence and $T_i$ donates the text in node $i$. $V_s$ indicates that every node in the sequence has the same contribution to the merged node.

$$V_s = \frac{1}{n} \cdot \sum_{i=1}^{n} MPNet(T_i) \tag{1}$$

The embedding vector of the merged node of an aggregation structure is shown in equation (2), where $k$ is the number of nodes in the aggregation structure, $n_i$ is the number of nodes merged in node $i$ (if $n_i \geq 2$, it is a sequence merged node, otherwise it is an original node), $T_{ij}$ is the text of index $j$ in node $i$.

$$V_a = \frac{1}{k} \cdot \sum_{i=1}^{k} (\frac{1}{n_i} \cdot \sum_{j=1}^{n_i} MPNet(T_{ij})) \tag{2}$$

An edge is denoted by a vector with two numbers, an entry node index and an exit node index, which indicates the direction of massage passing. The label is mapped into an one-hot vector to show the class the graph belongs to.

**Modeling.** It consists of two GNN layers, one pooling layer, one classifier layer, and the final predictive model.

(1) **GNN Layer.** The GNN layer is responsible for updating numerical representations of the graph, which generally includes two processing operations, message and aggregation, for graph nodes. Message deals with the feature information of graph nodes, and aggregation completes the aggregation operation of the information of its neighbor nodes for the current central node. The message and aggregation operations in different GNN models are different. The equation (3) is a general node feature aggregating process, where $i$ denotes the central node, $N$ is the number of neighbor nodes, message is the message operation (like LSTM, Transformer), $l$ is a layer number, $l+1$ is the next layer and $W$ is the weight of GNN layer, $x_j^l$ is the input vector of node $j$ to layer $l$. We create our model by adding two GNN layers: (1) the first layer accepts the initial input vector with 768 features and outputs a quadruple length (3072) vector, aiming to capture more fine-grained features. (2) the second layer reduces the dimensionality of the previous layer's output to 1024, making it easier to get the final output for prediction.

$$h_i^{l+1} = W \cdot \sum_{j=1}^{N} message(x_j^l) \tag{3}$$

(2) **Pooling Layer.** We use global soft attention layer [24] to reduce the size (length, width, number of channels) of the previous GNN layer, thereby reducing the amount of calculation, memory usage, and the number of parameters to achieve a certain scale, space invariance, and reduce the possibility of overfitting.

(3) **Classifier Layer and Prediction.** We create an 'argmax' function as a classifier to calculate the output vector, which consists of 0 and 1. The label where the index of '1' belongs to is the final graph predicted label.

## 4.2 GNN Models

In this paper, we consider six GNN-based models: Graph Convolutional Networks (GCNs) [20], Graph Attention Networks (GATs) [46], Unified Message Passing Model (UniMP) [44], GNNs with autoregressive moving average filter (ARMAConv) [2], Residual Gated Graph ConvNets (ResGatedGCNs) [3], and Feature-Steered Graph Convolutions (FeaStNet) [47], which represent various modeling strategies for dealing with graph representations.

The graph convolutional neural networks (GCNs) [20] is a feature extractor that operates on graphs. It is an aggregate operation on the Laplace matrix. The basic idea is that, for each node in a graph, its feature information is derived from all its neighbor nodes and its own features. Then, we apply appropriate functions, such as average, maximum, or more complex aggregate functions, to these features. Continually, we will perform the identical action on each node. The calculated features are then sent into the neural network.

Graph attention networks (GATs) [46] leverage the masked attention mechanism to aggregate the features of neighbor nodes in graphs to the central node. First, utilizing a linear mapping of shared parameters to increase the dimension of the feature of a

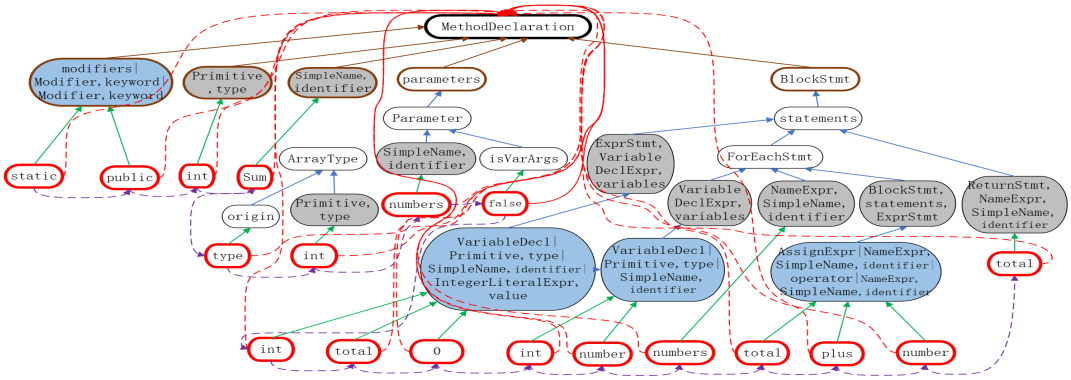



**Figure 5: A Sample CAG**

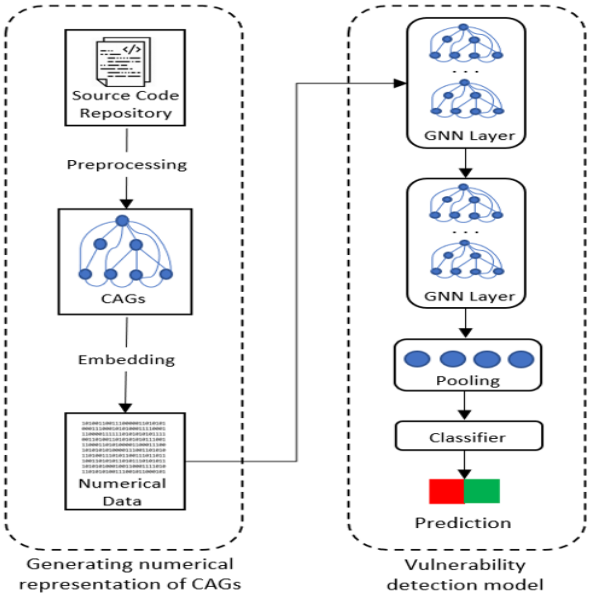



**Figure 6: Framework for GNN-based Vulnerability Detection**

node, and the transformed feature of the node is concatenated; Then, the concatenated high-dimensional feature is mapped to a real number and then using the softmax function to get the attention coefficient. Finally, the features are then weighted and summed in accordance with the attention coefficient to produce the new feature of the node.

Unified message passing model (UniMP) [44] predicts the label of a node in the graph based on the label information of the surrounding nodes. GCNs transform and propagate node features and predict the labels relying on the node features. UniMP uses both node features and label information in the training and prediction phases. In order to prevent information leakage caused by the use of label features during training, the model uses a mask model to hide some labels and then predicts the masked labels.

GNNs with auto-regressive moving average filter (ARMAConv) [2] uses the ARMA filter to improve the expressiveness of the model, capturing the global structure of the graph with few parameters. The residual gated graph convnets (ResGatedGCNs) [3] is a variant of GCNs by adding gated margins and residuals, which has a better performance on the semi-supervised clustering and subgraph matching problem. Feature-Steered Graph Convolutions (FeaStNet) [47] acquires network features through dynamic learning instead of relying on static coordinates.

## 5 EXPERIMENTS

The construction of AGs and CAGs of source code is based on 'eclipse.cdt.core' [12] for C and 'eclipse.jdt.core' [13] for Java. The six GNN models are implemented with Pytorch v1.11.0 [37] and PyG v2.0.1 [36] and executed on a multi-core server with 4 Tesla V100S-PCIE GPUs.

The performance metrics include accuracy, precision, recall, and F1 score. They are defined with respect to the numbers of true positive (TP), true negative (TN), false positive (FP), and false negative (FN). Accuracy is the percentage of total samples, including both positive and negative, that are predicted correctly, i.e., $(TP+TN)/(TP+TN+FP+FN)$. Precision measures that, among all samples that are predicted positive, how many of them are actually positive. For all the samples predicted as positive, there are two possible outcomes: TP and FP. Thus, precision is calculated by $TP/(TP+FP)$. Recall measures that, among all the positive samples in the dataset, how many of them are predicted as positive. It is defined as $TP/(TP+FN)$. F1 score is the harmonic mean of precision and recall, i.e., $2*(precision*recall)/(precision+recall)$.

We apply 5-fold cross-validation to the dataset in each experiment, 90% of the samples are used to build the classifier (including training and validation) and the remaining 10% to test the prediction model and report the results. The epoch of each experiment is initially set to 40. If the accuracy and loss values are satisfactory in the 40th epoch, the training is terminated, otherwise the training will continue. We report the geometric mean of the aforementioned evaluation metrics across 5 cross-validation folds.

### 5.1 The Datasets

We have compiled two datasets, one for Java and the other for C, as shown in Table 2. The Java dataset originates from three sources: the national vulnerability database (NVD), the software

**Table 2: The Datasets**

| Dataset | Language | #Vul. Types | #Positive Samples | #Negative Samples |
|---|---|---|---|---|
| Java Dataset | Java | 114 | 37,350 | 68,480 |
| C Dataset | C | 106 | 58,459 | 126,170 |

assurance reference dataset (SARD), and Luo et al.[4]. It has 37,350 positive samples with 114 vulnerability types and 68,480 negative samples. The C dataset, created from NVD and SARD, contains 58,459 positive samples with 106 vulnerability types and 126,170 negative samples. The above vulnerabilities have covered 37 of the 40 general categories of software vulnerability.

As mentioned in Luo et al. [30], the SARD samples have the following features that cause critical bias for machine learning: (1) the methods have logging statements before and after the vulnerable code to indicate the vulnerability location, (2) there are comments that describe vulnerability details, (3) many vulnerability samples use the same variable and method names. The avoid the effects of these features, we clean up the samples as follows: (1) removing all the comments. (2) removing the logging statements, (3) replacing the variable and method names recurring in the vulnerability samples with the names from non-vulnerable code.

## 5.2 GNN Models with CAGs

We have trained the six GNN models with the CAGs of the source code in the Java and C datasets. Tabel 3 presents the experiment results on the Java dataset. UniMP has achieved the best score on all performance indicators (accuracy, precision, recall, and F1). The 96.33% of accuracy and 96.08% of F1 score indicate UniMP has a good performance on both vulnerable and non-vulnerable samples. That is because it uses both node features and label information in the training and prediction phases help capture the vulnerability-related features. GATs' scores are close to UniMP (only 0.1% less on each indicator), so the attention mechanism is also helpful in capturing vulnerability features. GCNs' scores are 1% lower than UniMP. RGGCNs and ARMAConv, as new variants of GCNs, have slightly under-performed GCNs. The strategies of FeaStNet mostly applied to other domains and its performances are a bit lower than the other models. Table 4 shows the experiment result on C dataset. The evaluation scores of each model on C are 3% lower than the scores on Java. The main reason is that the syntax of Java in CAGs is more detailed than in C – Java CAGs carry more structural information. However, the overall performance of each model on C is still very good. UniMP remains the best among the six models.

Table 5 presents the experiment results of combining the Java and C datasets. The overall performance of each model is between those of the individual Java and C datasets. UniMP and GATs still have the best scores – 94.0% of accuracy and 93.4% of F1 score. The accuracy and recall are over 92%, and precision and F1 are over 91% of the other four models. Overall, the six models range from 91% to 96% on each performance indicator. This demonstrates that CAGs are an effective representation of source code for vulnerability detection with GNN models.

**Table 3: Results of the Java Dataset (%)**

| GNN Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| FeaStNet | 94.70 | 94.14 | 93.97 | 94.05 |
| ARMAConv | 95.01 | 95.35 | 94.02 | 94.14 |
| RGGCNs | 95.05 | 95.88 | 93.99 | 94.84 |
| GCNs | 95.30 | 95.60 | 94.22 | 94.91 |
| GATs | 96.27 | 96.50 | 95.04 | 95.84 |
| UniMP | **96.33** | **96.67** | **95.28** | **96.08** |

**Table 4: Experiment Results of the C Dataset (%)**

| GNN Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| FeaStNet | 91.55 | 90.77 | 92.14 | 91.29 |
| ARMAConv | 92.01 | 90.71 | 92.75 | 91.78 |
| RGGCNs | 92.49 | 90.79 | 92.98 | 91.91 |
| GCNs | 92.50 | 90.92 | 92.61 | 91.93 |
| GATs | 93.14 | **92.51** | 93.43 | 92.89 |
| UniMP | **93.21** | 92.48 | **93.47** | **92.90** |

**Table 5: Results of the Combined Java/C Dataset (%)**

| GNN Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| FeaStNet | 92.59 | 91.39 | 93.00 | 91.68 |
| ARMAConv | 92.75 | 91.72 | 92.05 | 91.88 |
| RGGCNs | 93.54 | 91.83 | 94.59 | 92.38 |
| GCNs | 93.49 | 91.39 | 93.74 | 92.43 |
| GATs | 94.07 | 92.28 | 94.72 | 93.39 |
| UniMP | **94.09** | **92.29** | **94.77** | **93.40** |

## 5.3 Effectiveness of Graph Reduction

To evaluate the usefulness of the reduction from AGs to CAGs, we have also applied the AGs of the datasets to the six GNN models. The AGs have 19 million nodes and 23.1 million edges. Training 10 epochs takes 31 minutes and 422 GB of numeric vector data. After the reduction, CAGs have 8.8 million and 12.9 million edges. In other words, 53.70% of the AG nodes and 44.23% of the AG edges are reduced. The vector data reduces to 190.3GB. The training time is deceased by 27.42%.

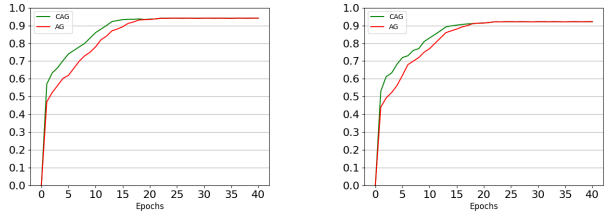
Figure 7 compares the result of CAGs and AGs on UniMP (the best performer of the six GNN models) of the Java dataset. After epoch 30, both models achieve optimum, and they have similar accuracy, precision, recall and F1 scores. The CAGs also make the learning faster for the same learning rate. Thus, CAGs are more efficient.

## 5.4 Comparison with ASTs, CFGs and PDGs

We have compared CAGs to ASTs, CFGs, and PDGs, which are among the most applied source code representations for program analysis. CFGs depict the flow of program execution, whereas PDGs capture data and control dependencies of code. Figure 8 shows the

**Table 6: Compression Ratios**

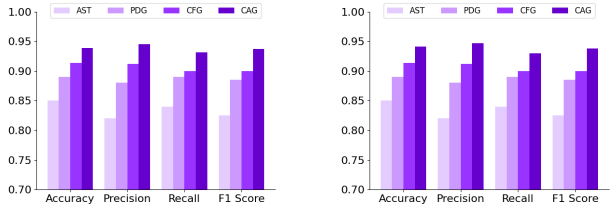
| Graph Representation | Node # | Node Reduction | Edge # | Edge Reduction | Time (min/10Epoch) | Training Time Reduction | Storage Size (GB) |
|---|---|---|---|---|---|---|---|
| AGs | 23,186,375 | - | 31,325,795 | - | 31.0 | - | 422 |
| CAGs | 10,746,584 | 53.65% | 18,885,202 | 39.71% | 22.5 | 27.42% | 190.3 |




(a) Accuracy (b) Precision

(c) Recall (d) F1 Score

**Figure 7: Comparison of CAGs and AGs**




(a) FeaStNet (b) ARMAConv

(c) RGGCNs (d) GCNs

(e) GATs (f) UniMP

**Figure 8: Comparison of Different Graph Representations**

experiment results. CAGs have significantly outperformed ASTs, CFGs, and PDGs. ASTs are poor for GNN-based vulnerability detection. All six GNN models have an average 84% accuracy and 81% F1 score. It indicates that the models are inefficient for both positive and negative samples. PDG-based models achieve an average 90% on both accuracy and F1 score, whereas CFG-based models got an average score of 92% on accuracy and F1 score.

## 5.5 Comparison with the Related Works

We have compared our approach with the recent related works: VulDeePecker [26], uVulDeePecker [56], Luo et al. [30], DEVIGN [55], Lin et al. [28], and FUNDED [48]. VulDeePecker and uVulDeePecker utilize BiLSTM on code segments. Luo et al. treats source code as text and checks integer overflow errors trough a fine-tuned BERT model. DEVIGN uses GNN models to learn features through an AST variant. FUNDED uses a GNN to operate on graph representation, which has multiple types of edges based-on ASTs and combines them through a GRU.

The above methods, except FUNDED, only work for C programs. Our comparative study uses the TIFS dataset from FUNDED [48], which has 38,845 negative and 34,035 positive samples. They account for 28 vulnerability types in C programs. Figure 10 shows the results of comparison. The UniMP model with AGs and CAGs has the best performance. GATs and GCNs with CAGs are slightly better than FUNDED, but much better than the other five models.

Figure 9 shows the performance of each method on individual vulnerability types. The BiLSTM-based models, VULDEEPECKER and uVULDEEPECKER, have performed well for CWE-190, CWE-191, and CWE-665, but their accuracy is less than 75% for other vulnerabilities. Luo's method focused on integer overflow errors (CWE-190). When applied to other vulnerabilities, its accuracy dropped to 70%. Even on CWE-190, UniMP with CAGs is 3% more accurate. Lin's method achieves over 80% accuracy for some vulnerabilities, 10 of which are over 90%, while UniMP with CAGs achieves the same or even better scores. DEVIGN using a standard GNN model achieves high accuracy only on several vulnerabilities. For example, DEVIGN's accuracy is only 63% for CWE-191, while UniMP with CAGs has a score of over 91%. For most vulnerability types, FUNDED has an average score of over 85%; however, it still achieves a low accuracy in detecting CWE-191, CWE-400, and CWE-404. The UniMP model with AGs and CAGs achieved over 90% accuracy on 27 of the 28 vulnerability types. For seven

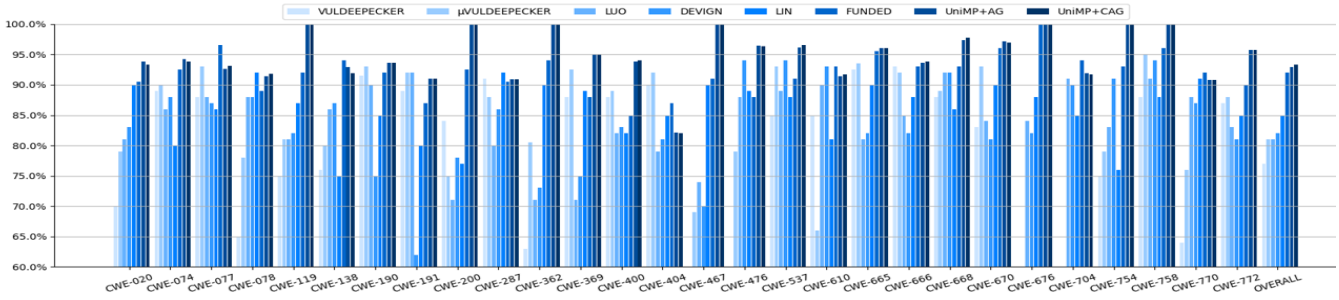



**Figure 9: Comparison of Individual Vulnerabilities in the TIFS Dataset**

**Table 7: The Applications**

| Project | Lan. | KLOC | Classes | Methods | #Vul. |
|---|---|---|---|---|---|
| Apache JMeter | Java | 122 | 1,921 | 9,306 | 160 |
| Elasticsearch | Java | 366 | 4,836 | 26,646 | 405 |
| FFmpeg | C | 615 | 3,478 | 36,505 | 627 |
| OpenSSL | C | 361 | 2,203 | 19,922 | 630 |
| GitHub | J&C | - | - | 1,506 | 1,506 |
| Total | - | 1,464 | 12,438 | 93,132 | 2,575 |

vulnerability types, the accuracy is even 100%. The accuracy of CWE-404 is 82%, the only one below 90%.

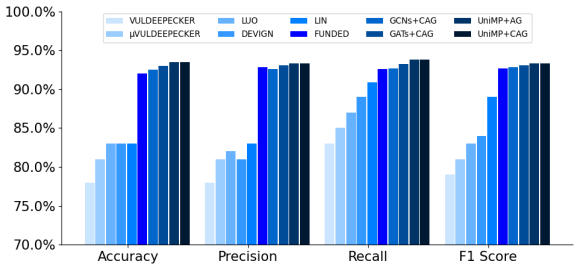



**Figure 10: Comparison Result on TIFS Dataset**

## 6 APPLICATIONS

We have applied the resultant GNN models trained with the CAGs (Section 5.2) to four open-source projects and many GitHub projects with historic vulnerabilities. Table 7 shows the list of applications. The project sizes range from 122 KLOC (thousand lines of code) to 615 KLOC. They have a total of 1,464 KLOC, 12,438 classes, and 92,379 methods. We also extracted 1,506 vulnerable routines from GitHub projects according to their commits data using a pre-trained expert model [29]. In total, there are 2,575 vulnerable routines.

Table 8 shows the prediction results. All six GNN models have achieved high scores on accuracy (from 93.3% to 96.9%) and precision (from 87.4% to 94.3%). UniMP's recall and F1 scores, 88.5% and 91.3%, are significantly higher than the other models. The false positive rates (FPRs) of the models are similar. However, UniMP's false negative rate (FNR) is much lower than the other models. Compared to the training results in Tables 3, 4, and 5, the performances for the real-world applications have dropped but remain promising. In particular, the UniMP model is outstanding.

**Table 8: Prediction Results for Real-World Applications (%)**

| GNN Model | A | P | R | F1 | FPR | FNR |
|---|---|---|---|---|---|---|
| FeaStNet | 93.3 | 87.4 | 74.6 | 80.5 | 2.5 | 25.4 |
| ARMAConv | 93.3 | 91.1 | 70.7 | 79.6 | 1.6 | 29.3 |
| RGGCNs | 93.3 | 90.3 | 70.8 | 79.4 | 1.7 | 29.2 |
| GCNs | 95.1 | 96.9 | 76.0 | 85.2 | 0.6 | 24.0 |
| GATs | 95.3 | 94.6 | 79.2 | 86.2 | 1.0 | 20.8 |
| UniMP | 96.9 | 94.3 | 88.5 | 91.3 | 1.2 | 11.5 |

## 7 CONCLUSIONS

We have presented CAGs as a novel source code representation for predicting software vulnerabilities with GNN models. The experiments using two large Java and C datasets with 220 types of vulnerabilities have demonstrated that CAGs are much more efficient than ASTs, CFGs, and PDGs, which are among the widely applied source code representations. The resultant GNN models have achieved promising performance when applied to more than 2,500 vulnerabilities collected from real-world software projects. The comparative study has also shown that the CAG-based GNN models can outperform the existing machine-learning based methods for vulnerability detection.

In this paper, CAGs are applied to Java and C programs. As a language-agnostic representation of source code, they can be used for different programming languages. Our future work will expand this paper to detect vulnerabilities in other popular languages, such as C#, C++, and Python.

## ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation (NSF) under grants 1820685 and 2101118. Ruoyao Xiao assisted in the control flow graph (CFG) experiment.

# REFERENCES


[1] M. Allamanis, M. Brockschmidt, and M. Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

[2] F.M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. 2021. Graph neural networks with convolutional arma filters. *IEEE transactions on pattern analysis and machine intelligence*.

[3] X. Bresson and T. Laurent. 2017. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*.

[4] Checkmarx. 2006. Checkmarx. https://www.checkmarx.com. (2006).

[5] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie. 2009. Brick: a binary tool for run-time detecting and locating integer-based vulnerability. In *2009 International Conference on Availability, Reliability and Security*. IEEE, 208–215.

[6] R. Chinchani, A. Iyer, B. Jayaraman, and S Upadhyaya. 2004. Archerr: runtime environment driven program safety, 385–406.

[7] M. Choi, S. Jeong, H. Oh, and J. Choo. 2017. End-to-end prediction of buffer overruns from raw source code via neural memory networks. *arXiv preprint arXiv:1703.02458*.

[8] [n. d.] Common vulnerabilities and exposures. https://cve.mitre.org. ().

[9] H.K. Dam, T. Tran, T. Pham, S.W. Ng, J. Grundy, and A. Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*.

[10] J. Devlin, M. Chang, K. Lee, and K. Toutanova. 2018. Bert: pre-training of deep bidirectional transformers for language understanding. (2018). arXiv: 1810.04805 [cs.CL].

[11] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang. 2018. Defect prediction in android binary executables using deep neural network. *Wireless Personal Communications*, 102, 3, 2261–2285.

[12] [n. d.] Eclipse cdt. https://wiki.eclipse.org/Getting_started_with_CDT_development. ().

[13] [n. d.] Eclipse jdt. https://www.eclipse.org/jdt/core/. ().

[14] Z. Feng et al. 2020. Codebert: a pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

[15] Flawfinder. [n. d.] Flawfinder. http://www.dwheeler.com/flawfinder. ().

[16] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun. 2018. Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 896–899.

[17] P. Godefroid, M. Levin, and D. Molnar. 2008. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*.

[18] G. Holmes, A. Donkin, and I.H. Witten. 1994. Weka: a machine learning workbench. In *Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference*. IEEE, 357–361.

[19] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. 2012. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, 7–10.

[20] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

[21] D. Kravets. 2017. Sorry ma'am you didn't win $43m – there was a slot machine 'malfunction'. Ars Technica, editor. (June 2017). https://arstechnica.com/tech-policy/2017/06/sorry-maam-you-didnt-win-43m-there-was-a-slot-machine-malfunction/.

[22] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu. 2018. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *International Conference on Learning Representations*.

[23] Y. Lee, S. Choi, C. Kim, S. Lim, and K. Park. 2017. Learning binary code with deep learning to detect software weakness. In *KSII the 9th international conference on internet (ICONI) 2017 symposium*.

[24] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

[25] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. 2021. Sysevr: a framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*.

[26] Z. Li, D. Zou, S. Xu, X. Ou, and Y. Zhong. 2018. Vuldeepecker: a deep learning-based system for vulnerability detection. In *Network and Distributed System Security Symposium*. (Feb. 2018).

[27] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang. 2017. Poster: vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2539–2541.

[28] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague. 2018. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics*, 14, 7, 3289–3297.

[29] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*.

[30] Y. Luo, W. Xu, and D. Xu. 2021. Detecting integer overflow errors in java source code via machine learning. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 724–728.

[31] Microsoft. 2012. Prefast analysis tool. https://msdn.microsoft.com/en-us/library/ms933794.aspx. (2012).

[32] S. A Mokhov, J. Paquet, and M. Debbabi. 2015. Marfcat: fast code analysis for defects and vulnerabilities. In *2015 IEEE 1st International Workshop on Software Analytics*. IEEE, 35–38.

[33] [n. d.] National vulnerability database. https://nvd.nist.gov. ().

[34] Y. Pang, X. Xue, and A. Namin. 2015. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 543–548.

[35] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. 2015. Building program vector representations for deep learning. In *International conference on knowledge science, engineering and management*. Springer, 547–553.

[36] [n. d.] Pyg. https://pytorch-geometric.readthedocs.io/en/latest/. ().

[37] [n. d.] Pytorch. https://pytorch.org/. ().

[38] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.

[39] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes. 2016. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, 1157–1168.

[40] SARD. 2019. Nist software assurance reference dataset project. https://samate.nist.gov/SRD/testsuite.php. (2019).

[41] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40, 10, 993–1006.

[42] C.D. Sestili, W.S. Snavely, and N.M. VanHoudnos. 2018. Towards security defect prediction with ai. *arXiv preprint arXiv:1808.09897*.

[43] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. 2009. Detection of malicious code by applying machine learning classifiers on static features: a state-of-the-art survey. *information security technical report*, 14, 1, 16–29.

[44] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjin Wang, and Yu Sun. 2020. Masked label prediction: unified message passing model for semi-supervised classification. *arXiv preprint arXiv:2009.03509*.

[45] K. Song, X. Tan, T. Qin, J. Lu, and T. Liu. 2020. Mpnet: masked and permuted pre-training for language understanding. *Advances in Neural Information Processing Systems*, 33, 16857–16867.

[46] Petar V., Guillem C., Arantxa C., Adriana R., Pietro L., and Yoshua B. 2018. Graph attention networks. In *International Conference on Learning Representations*.

[47] N. Verma, E. Boyer, and J. Verbeek. 2018. Feastnet: feature-steered graph convolutions for 3d shape analysis. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2598–2606.

[48] H. Wang, G. Ye, Z. Tang, S.H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE TIFS*, 16, 1943–1958.

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

[50] T. Wang, T. Wei, Z. Lin, and W. Zou. 2009. Intscope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network & Distributed System Security Symposium (NDSS2009)*.

[51] F. Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[52] R. Wojtczuk. 2005. Uqbtng: a tool capable of automatically nding integer overows in win32 binaries. *Proceedings of Chaos Communication Congress*.

[53] F. Yamaguchi, F. Lindner, and K. Rieck. 2011. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*, 13–13.

[54] G. Zhao and J. Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 141–151.

[55] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

[56] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. 2019. $\mu$Vuldeepecker: a deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*.