Received 20 September 2022, accepted 12 October 2022, date of publication 21 October 2022, date of current version 30 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3216395

RESEARCH ARTICLE

Code Aggregate Graph: Effective Representation for Graph Neural Networks to Detect Vulnerable Code

HOANG VIET NGUYEN[1], JUNJUN ZHENG[2], (Member, IEEE), ATSUO INOMATA[2], (Member, IEEE), AND TETSUTARO UEHARA[1], (Member, IEEE)
[1]College of Information Science and Engineering, Ritsumeikan University, Kusatsu 5258577, Japan
[2]Graduate School of Information Science and Technology, Osaka University, Osaka 5650871, Japan

Corresponding author: Hoang Viet Nguyen (hoang@cysec.cs.ritsumei.ac.jp)

ABSTRACT Deep learning, especially graph neural networks (GNNs), provides efficient, fast, and automated methods to detect vulnerable code. However, the accuracy could be improved as previous studies were limited by existing code representations. Additionally, the diversity of embedding techniques and GNN models can make selecting the appropriate method challenging. Herein we propose Code Aggregate Graph (CAG) to improve vulnerability detection efficiency. CAG combines the principles of different code analyses such as abstract syntax tree, control flow graph, and program dependence graph with dominator and post-dominator trees. This extensive representation empowers deep graph networks for enhanced classification. We also implement different data encoding methods and neural networks to provide a multidimensional view of the system performance. Specifically, three word embedding approaches and three deep GNNs are utilized to build classifiers. Then CAG is evaluated using two datasets: a real-world open-source dataset and the software assurance reference dataset. CAG is also compared with seven state-of-the-art methods and six classic representations. CAG shows the best performance. Compared to previous studies, CAG has an increased accuracy (5.4%) and F1-score (5.1%). Additionally, experiments confirm that encoding has a positive impact on accuracy (4–6%) but the network type does not. The study should contribute to a meaningful benchmark for future research on code representations, data encoding, and GNNs.

INDEX TERMS Vulnerability detection, code representation, graph neural networks, deep learning.

I. INTRODUCTION

Vulnerabilities in software programs are escalating. For example, the number of vulnerabilities in open-source software has almost doubled from more than 6,000 vulnerabilities in 2019 to nearly 10,000 vulnerabilities in 2020 [1]. Vulnerabilities impact both developers and users. In 2021, the CVE-2021-44832 vulnerability in open-source software Log4j caused over 800,000 exploitation attempts in a three-day period and affected 95% of Java programs [2]. Hence, more effective solutions to detect security vulnerabilities, especially in the early development process, are needed.

Many methods have been proposed to detect software security vulnerabilities. From the perspective of source code execution, these methods can be divided into two categories: dynamic and static detection. Dynamic vulnerability detection examines vulnerabilities when software is executed and observes its behaviors. Examples include fault injection and fuzzy testing. In contrast, static detection is independent of code execution and is useful in terms of code coverage and handling with diverse vulnerabilities [3]. Nevertheless, traditional static detection methods such as rule-based ones have low accuracies and high false positive rates [4]. A promising approach to increase the accuracy with less human intervention is to apply machine learning to static methods [5], [6], [7], [8]. However, human experts must define rules or features to generate vulnerability patterns, which are burdensome. Additionally, accurately characterizing vulnerabilities is a difficult task. Hence, machine learning often generates many false positives and false negatives.

The associate editor coordinating the review of this manuscript and approving it for publication was Mehedi Masud.

Recently, deep learning (DL) methods have received attention as they can detect vulnerability patterns automatically. Fundamentally, DL methods learn syntax or semantic properties of the source code via suitable representations. These representations are often vectorized into forms that computers can easily understand and extract information. A deep model can also classify whether the source code is vulnerable. A common approach is to treat the source code as a special type of text and analyze it utilizing natural language processing (NLP) [6], [7]. Because source code requires stricter grammar and more robust logic than natural language, treating source code as a sequential feature representation may limit DL models' potential. A few studies have represented the source code in a more logical and structured manner such as vectors or graphs [8], [9], [10]. These proposals have demonstrated that adopting graph representations improves the effectiveness of DL in identifying liable code.

Although graph-based representation combined with DL is a suitable approach, it faces many challenges. A critical one is the need to improve the detection performance. Recent approaches have achieved an accuracy of 75% for real-world source code [8]. Although graph representation outperforms traditional methods, its accuracy is inferior to human performance. One limitation of DL is that current graph representations cannot capture complexity, which is needed to help a model learn the correct code structure. Incorporating a dominator tree (DT) and a post-dominator tree (PDT) into vulnerable code detection is promising for several types of vulnerabilities [11]. Here, we combine DT and PDT with other graph representations to increase the ability to capture the semantic information of the source code, and consequently increase the vulnerability detection efficiency.

Another issue is that it is unclear which factors (e.g., word embedding or GNN model) influence the effectiveness. Although many different embedding methods and GNNs have been proposed, previous studies have not evaluated their impact on the detection performance. This is a significant shortcoming because word embedding directly affects the data multiplication of the graph, while GNN models act as classifiers to extract and read that data.

To address these issues, we develop a code graph representation called Code Aggregate Graph (CAG) to improve efficiency and provide a comprehensive evaluation of the factors affecting detection ability. We also implement a code graph built based on DT and PDT to improve performance and compare CAG to traditional techniques. Finally, we examine the influence of word embedding and GNNs, which have been previously overlooked. These results should provide a reference baseline for future comparison studies.

The main contributions of this paper are as follows:

- Development of a new graph representation, which leverages the advantages of DT and PDT to improve the effectiveness of deep graph network in detecting vulnerable code;
- Investigation of the influence of word embedding methods and graph network models on the detection performance;
- Comparison of the vulnerability detection performance between six popular graph representations for C code;
- Validation of the effectiveness of CAG using two different datasets.

The remainder of this paper is organized as follows. Section II reviews related works on vulnerability detection. Section III describes CAG, including the related code graph representation and its implementation. Section IV discusses the embedding methods and GNNs that are used. Section V presents our main research questions, datasets, and experiment setup. Section VI details the proposed method being tested. Section VII considers the limitations of this study. Finally, Section VIII concludes this paper.

II. RELATED WORK

Diverse techniques have been proposed to detect bugs and vulnerabilities. Examples include metric-based, pattern-based, and binary-based methods. Here, we discuss techniques directly related to our proposed method.

Techniques using code as the main basis for analysis can be divided into two types: code similarity-based and pattern-based. Code similarity approaches find matches between target codes with known vulnerabilities for classification. VDSimilar [12] and VUDDY [13] are two examples. Although VDSimilar uses a Siamese network [14] along with BiLSTM to improve the detection accuracy, VUDDY improves the scalability of vulnerable code clone detection using function-level granularity and a length-filtering technique.

On the other hand, pattern-based approaches comprehend the properties of source code and use them to check the target code. There are two sub-categories of pattern-based methods: rule-based and machine learning-based. In rule-based approaches, vulnerability patterns created by experts are used to detect vulnerabilities. Approaches in this sub-category, namely RATS [15] and Flawfinder [16], are efficient for specific types of vulnerabilities such as stack-based buffer overflow, heap-based buffer overflow, and format string vulnerabilities [17]. Machine learning-based approaches use patterns from code graph representations such as abstract syntax tree (AST) or control flow graphs (CFG), in cooperation with traditional machine learning approaches, namely support vector machines [18], logistic regression [19], and decision tree [20], to detect vulnerabilities. These approaches partly depend on manually generated data sources.

DL methods are widely used for vulnerability detection. Leveraging the power of deep learning can help address the manual pattern problem. Inspired by NLP, Lin et al. [6] proposed a benchmark framework with six different neural networks to detect vulnerable code functions. VulDeePecker [9] converts programs into vectors and then extracts features from graph representations at the slice level. SySeVR [10] extends the idea of VulDeePecker to detect multiclass vulnerabilities.

Although these vector extraction approaches can learn code features, some information is lost during the transformation process. Cao et al. [8] utilized AST and CFG graphs along with GNNs to learn code features. They also suggested using both forward and backward edges to increase the vulnerability detection accuracy. Although initial results have been achieved, the vulnerability detection can be improved. Since deep neural networks and graph representations were together, the performance of each component should be evaluated for a better understanding. Our proposed method, CAG, was created to overcome these limitations. by implementing a novel graph, CAG improves the accuracy of vulnerability detection while simultaneously evaluating the impact of each component on the detection efficiency.

III. CODE AGGREGATE GRAPH

This research aims to improve the vulnerability detection performance by applying a representation, which can more accurately depict the semantic information of the algorithm. CAG is designed to enhance the ability to represent syntactic and semantic information of the source code. This section initially discusses the phenomena motivating the idea of CAG. Then details of the implementation and algorithms used to construct the graph are explained.

A. OBSERVATIONS

Analyzing the source code input data revealed several phenomena. To demonstrate these phenomena, here a dummy function named *foo* is introduced. The function *foo* simply checks whether an input from a user is greater than a given constant *threshold* value. The function returns a valid value and sends that value to another function for processing. The function *foo* contains an error (bug) because parameter correctness checking before calling is omitted. After this bug was discovered, the developers patched the function. Figure 1 shows details of the vulnerable and patched version of *foo*. In addition, Fig. 2 shows the control flow and DT of these versions.

```
void foo() {
    int x = input();
    int y;
    if (x > THRESHOLD) {
        y = 0;
    } else {
        y = x;
    }
    send(x);
    return y;
}
```

(a) Function $send(x)$ contains vulnerable code as there is not a condition to check for out-of-bounds before calling.

```
void foo() {
    int x = input();
    int y;
    if (x > THRESHOLD) {
        y = 0;
    } else {
        y = x;
        send(x);
    }
    return y;
}
```

(b) Modified patched code only calls the $send(x)$ function when x is guaranteed to be less than the threshold value.

FIGURE 1. Code example.

1) FIRST OBSERVATION

In the CFG of the vulnerable code diagram (Fig. 2a), the command $send(x)$ indicates that it is directly associated with the statement $y = 0$ or $y = x$. Although these two commands are not the root cause of the bug, the control flow graph highlights their immediate relation. On the other hand, the dominator tree (Fig. 2b) shows that this defect only arises from previous statements, especially the line $x > threshold$. In addition, it demonstrates that the defect is unrelated to $y = 0$ or $y = x$. In this case, DT explains the reason for the vulnerability better than CFG. Although DT is generated from CFG, their purposes differ. CFG describes what can happen, while DT describes the ordinal nature of the statement.

This phenomenon has appeared in real-world vulnerable codes such as CVE-2020-35605, CVE-2021-38383, and CVE-2021-38206. In CVE-2020-35605 (Fig. 3b), the bug occurred in the graphics protocol feature of the *graphics.c* file, and allowed an attacker to execute arbitrary code. This bug, which is in line 405, calls function *ABRT* and uses the filename (*fname* variable) as a parameter for the function. The bug is due to the absence of a filename filter, which may have special characters. Although the control flow "tie" statement contains this bug with the two open file statements (line 403 and 404), the bug originates from the variable declaration at the beginning of the function. In this case, DT also provides a more accurate view of the source of the defect than CFG.

Previous studies have demonstrated that DT can be applied to detect vulnerable code. Schafer et al. [11] showed that DT can effectively detect code clone vulnerabilities. Despite using a statistical approach for a specific type of vulnerability, their paper also serves as proof of the untapped potential of DT for characterizing source code. Therefore, we integrated DT into CAG to enhance its performance.

In addition, Cao et al. [8] confirmed that adding bidirectional edges increases the ability to recognize vulnerabilities. Instead of building another direction for all the edges in CAG, we used PDT to represent the relationship between the nodes with the correlation to the end node.

2) SECOND OBSERVATION

Figure 1b shows the patch for the example in Fig. 1a, where the developers moved the order of execution of command lines to fix the unsafe code *send(x)*. The line is moved from outside to inside the conditional statement. If only AST is used, the bug is undetectable because AST helps recognize changes in the syntax. However, in this case, both the

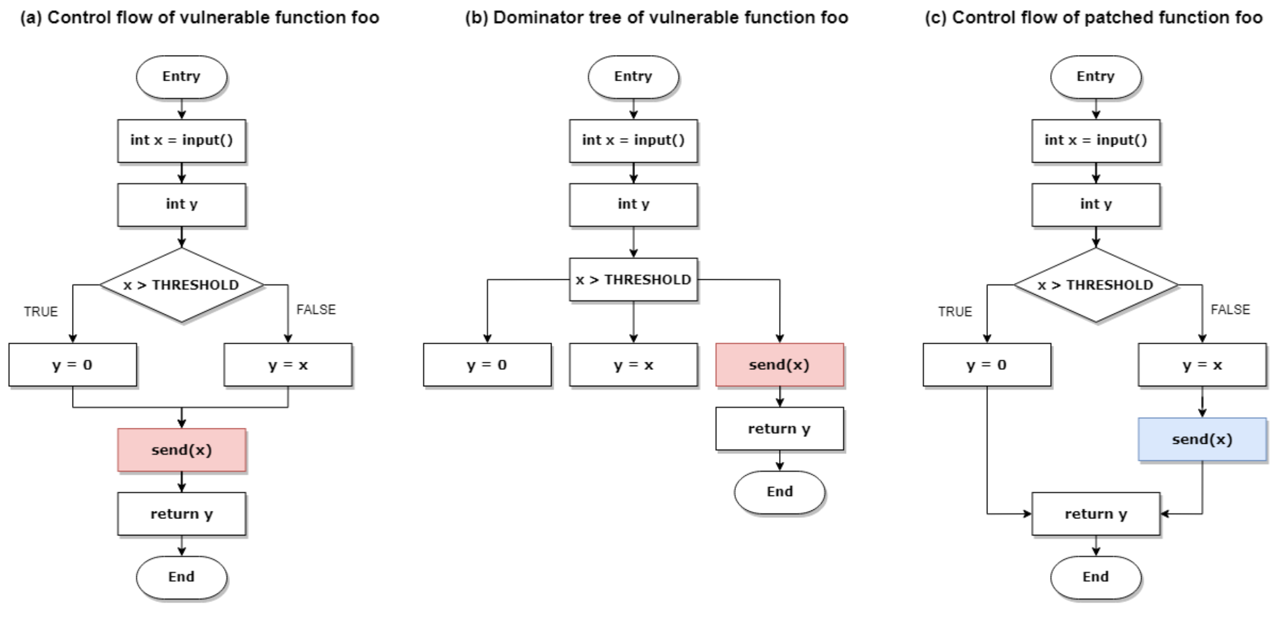
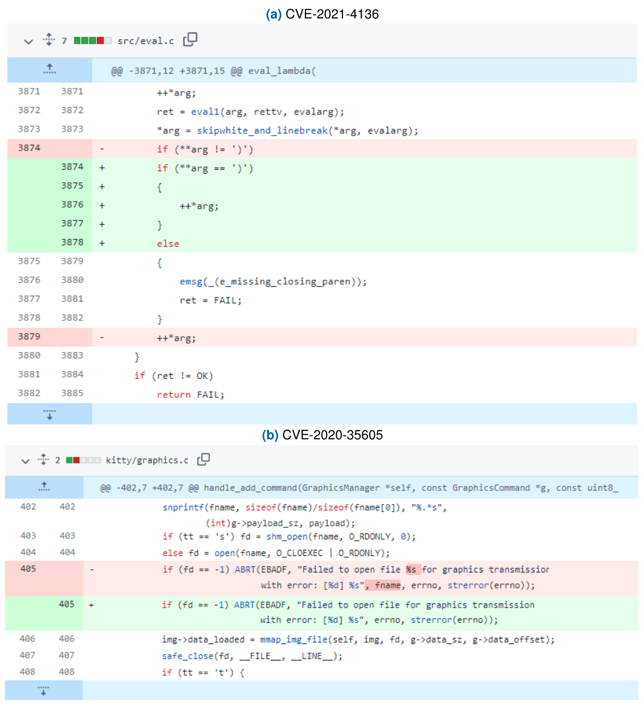


FIGURE 2. Control flow and dominator tree for vulnerable and patched versions of function *foo*.


(a) CVE-2021-4136

```
7  src/eval.c
@@ -3871,12 +3871,15 @@ eval_lambda(
3871 3871          ++*arg;
3872 3872          ret = eval1(arg, rettv, evalarg);
3873 3873          *arg = skipwhite_and_linebreak(*arg, evalarg);
3874      -         if (**arg != ')')
     3874 +         if (**arg == ')')
     3875 +         {
     3876 +             ++*arg;
     3877 +         }
     3878 +         else
3875 3879          {
3876 3880              emsg(_(e_missing_closing_paren));
3877 3881              ret = FAIL;
3878 3882          }
3879      -         ++*arg;
3880 3883      }
3881 3884      if (ret != OK)
3882 3885          return FAIL;
```

(b) CVE-2020-35605

```
2  kitty/graphics.c
@@ -402,7 +402,7 @@ handle_add_command(GraphicsManager *self, const GraphicsCommand *g, const uint8_
402 402            snprintf(fname, sizeof(fname)/sizeof(fname[0]), "%.*s",
                           (int)g->payload_sz, payload);
403 403            if (tt == 's') fd = shm_open(fname, O_RDONLY, 0);
404 404            else fd = open(fname, O_CLOEXEC | O_RDONLY);
405      -         if (fd == -1) ABRT(EBADF, "Failed to open file %s for graphics transmissior
                                      with error: [%d] %s", fname, errno, strerror(errno));
    405 +         if (fd == -1) ABRT(EBADF, "Failed to open file for graphics transmission
                                      with error: [%d] %s", errno, strerror(errno));
406 406            img->data_loaded = mmap_img_file(self, img, fd, g->data_sz, g->data_offset);
407 407            safe_close(fd, __FILE__, __LINE__);
408 408            if (tt == 't') {
```

FIGURE 3. Real-world vulnerability examples.

vulnerable code and the patch functions contain the same syntax. The difference between the two is the control flow of the function.

Real-world scenarios contain a lot of similar cases such as in vulnerabilities CVE-2021-4136 (Fig. 3a). This vulnerability is exploited to use against the Heap-based Buffer Overflow due to the lack of *arg* variable checking. The given solution moves the command line 3879 (++**arg*) inside the main conditional statement. Although the text does not change significantly, this vulnerability contains a change in the control flow of the function. A similar manifestation has also been recorded in the CVE-2019-190753 vulnerability [8].

This observation shows that both syntax and semantics are necessary to fully capture the source code characteristics. Hence, more than one type of graph should be utilized. Here, we combine AST, CFG, program dependence graph (PDG) and two types of trees, DT and PDT, to ensure that the models can observe the attributes of the source code.

B. IMPLEMENTATION

As discussed above, each graph representation can capture different features of the provided source code. Utilizing these features individually is insufficient to distinguish normal functions from vulnerable ones. Therefore, a suitable method is needed to combine different types of representations into a unified whole. In addition, the use of DT and PDT can realize advantages for the detection ability. In this study, we propose CAG, which is built based on an AST, CFG, PDG, DT, and PDT.

Below, each type of graph representation and its definition are introduced in the form of joint representation. Then the approach to extract the immediate DT and immediate PDT is portrayed with a simple and direct version of the original one. Finally, the algorithm used to construct CAG is discussed in detail.

1) AST

AST is a common tree representation used to analyze the syntactic structure of source code [21]. It can precisely represent how a program code is constructed. AST is an important kernel in the semantic analysis phase of compilers [22]. AST is composed of operators (non-leaf nodes) and operands (leaf nodes). Although semantically similar code can be identified, AST is generally unavailable for further analysis since it does not represent the control flow or data dependencies. Figure 4 provides an example of AST for the function *foo*.

We use a joint representation by describing AST as $G_{AST} = (V, E, C)$, where $V = \{v_1, \ldots, v_n\}$ represents the set of nodes in the graph, and n is the number of nodes. The set $E = \{e_1, \ldots, e_k\}$ is the edges in AST, where k is the number of edges of tree, and $e_i = (\gamma_i)$ contains edge type information. $C = \{c_1, .., c_n\}$ represents source code data in each node.

2) CFG

CFG analyzes the execution order of code statements [23]. It considers both structured and unstructured control statements such as if, else, for, or goto. The edges of CFG can be assigned one of three labels: true, false, or ϵ. Non-control statements are labeled as ϵ, while control statements are assigned as true or false, depending on the case. Although CFG can represent the source code structure, it cannot show the data flow, which is often exploited by attackers.

CFG is represented by $G_{CFG} = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of nodes representing for statements and predicates in AST. $E = \{e_1, \ldots, e_k\}$ is the set of edges with a size of k. Each edge $e_i = (\gamma_i, \lambda_i)$ where γ is the edge type and λ contains label information to give the true, false or ϵ value.

3) PDG

PDG is an intermediate program representation that may expose the operation dependencies for a program [24]. PDG includes two types of edges: data dependency edges and control dependency edges. The former indicates the effect of one variable on another, while the latter reflects the influence of predicates on the variable values [25]. Control dependency edges are often used to explicitly represent the relationship between statements and predicates. Hence, the nodes of this representation are identical to those of CFG, but their edge properties differ. In PDG, an edge represents control dependencies or data dependencies.

4) DT AND PDT

If CFG only shows the possible order of the statements, DT indicates the order in which they must happen [26]. Due to this certainty, DT is often used to optimize compilers. First, we explain the concepts of domination and post-dominate. Node x is said to dominate node y if and only if all paths from the entry node to y go through x. Otherwise, node x is a post-dominate node of y when all paths from the end node to y go through x.

The immediate DT is a shortened version of DT, but retains the properties of DT. Node x is considered the immediate dominator of node y if and only if x dominates y and all of y's dominators dominate x. Here, Algorithm 1 is used to build an immediate DT from the dominator relationship between two nodes.

Algorithm 1 DT Creation From the Dominator Relation

```
Input: Node set $V = \{v_1, v_2, \ldots, v_t\}$ Function $dom(v_i)$
        defines nodes being nominated by $v_i$
Output: Immediate dominator edge set
        $E = \{e_1, e_2, \ldots, e_k\}$ $e_i = (u, v), \forall u, v \in V$
Initialize E empty;
for $v \in V$ do
    $dominates_1 = dom(v)$ ;
    for $u \in V, \forall u \in dominates_1$ do
        $dominates_2 = dom(u)$ ;
        if $i \in dominates_2, \forall i \in dominates_1$ then
            $e = create\_edge(u, v)$ {u immediately
              dominates v};
            $E.insert\_edge(e)$ ;
return $E$;
```

We use the immediate DT and immediate PDT to ensure the representation of the relationship between nodes and maintain a suitable number of edges, which can be handled by the training models. For ease, we also use the concepts of DT and PDT to call their immediate derivatives.

5) BUILDING CAG

CAG is the synthesis of different graph representations into a unified one. Inheriting the advantages of the above graphs, CAG can express the semantic and structural properties as well as show the control flow and data flow of the source code. Figure 5 overviews the graph for the function *foo*.

Algorithm 2 CAG Implementation

```
Input: Edge type set $T = \{AST, CFG, PDG, DT, PDT\}$
        Graphs $G_i = (V_i, E_i, C_i)$; $i \in T$
Output: Aggregate representation
        $\{z_v^e, \forall v \in V_{AST}, \forall e \in E_i\}$
Initialize Z empty;
for $v \in V_{AST}$ do
    $\zeta \leftarrow extract\_node\_data(v)$ {Get source code and
      type of node v};
    $\epsilon_i \leftarrow get\_edges(v); i \in T$ {Get all edges of node v};
    for $e \in \epsilon_i$ do
        $\gamma \leftarrow extract\_label(e)$ {Get edge type};
        $Z.insert(v, e, \gamma, \zeta)$;
return $Z$;
```

Algorithm 2 describes the process of constructing CAG. The input is all graph representations defined in the form of joint representation $G = (V, E, C)$, where V and E are the nodes and edges of the graph, respectively. C is the code

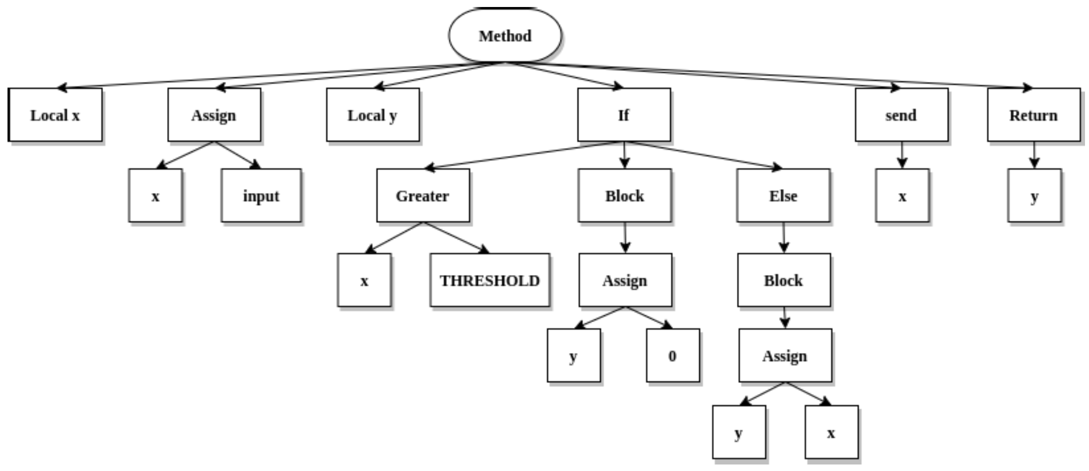


FIGURE 4. Example of AST for the function *foo*.

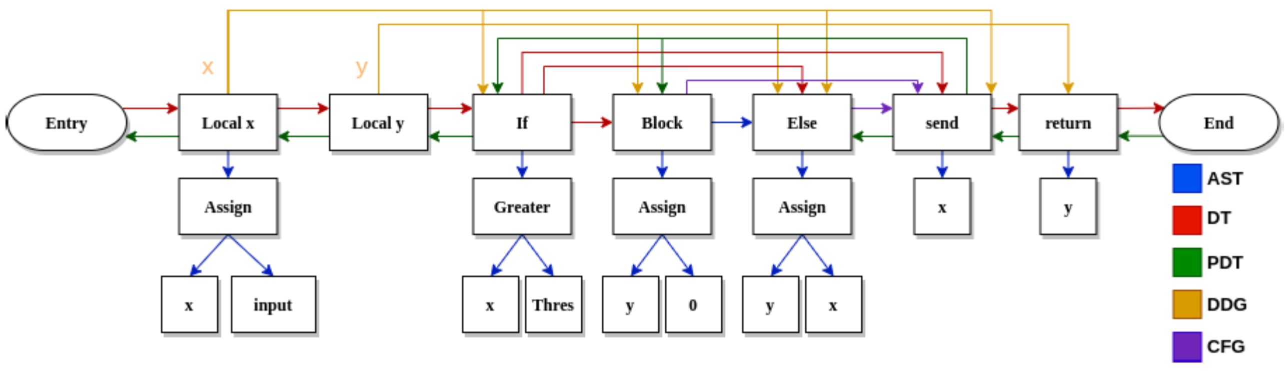


FIGURE 5. Example of CAG for the function *foo*.

statement data. Because AST is the only representation that covers all the nodes in the graph, CAG is constructed based on traversing each node *v* of AST. First, the node types and source code statements are extracted from each node through the function *extract_node_data*. If the node does not contain code, for example, the BLOCK node, empty data is returned. These pieces of information are necessary for data encoding (Section IV-B). Then the edges existing on each node are retrieved for all types of graph representations through the *get_edges* function. At each edge in the resulting set, the function *extract_label* is called to obtain information about the edge type and additional information. For example, additional information on PDG is the name of the variable. Finally, the node and edge information are aggregated and assigned to the new graph. The obtained graph after traversing all the nodes is CAG.

IV. EMBEDDINGS AND MODELS

A. OVERVIEW

Although the code representation greatly impacts the efficiency of identifying vulnerable code, other factors also affect the performance. Because the source code information is encoded based on NLP, choosing the proper word embedding plays an important role in precisely capturing the code's semantic data. Previous studies have revealed that word embedding significantly influences the detection performance in text-based approaches [7], [27]. Another essential factor is deep GNNs. Network models are primarily responsible for distinguishing between normal and vulnerable codes. In the context of the code graph representation, the syntactic aspect of the source code is useful. Therefore, the impact of GNNs should be considered when evaluating the overall efficiency.

Although word embedding and GNN models are two influential factors, previous studies have not investigated their impact on the overall results of the code graph representation-based technique in detail. Devign [28] and BGNN4D [8] both use Word2Vec embedding with a GNN model. They employ the gated graph recurrent model. Although both the word embedding and neural networks can affect detection accuracy, their effects were not mentioned. Herein, we evaluate the contribution of different embedding approaches and models on the performance to create a benchmark that can be used for future studies.

B. DATA ENCODING

Data encoding transforms data in a code graph representation into vectors, which are suitable for neural networks to learn. After converting into a code graph representation, the source code is in the form of connected nodes. Each node usually represents a type and a statement of the code. The data encoding process turns the information on each node (the code statement and node type) into a vector using different word embeddings. The process is essential for our system to recognize the similarities between nodes as well as to learn the semantic information from the source code.

Here, we utilize popular world embedding methods such as Word2Vec, Glove, and FastText. Word2Vec is a fundamental technique in NLP due to its effectiveness and applicability. Word2Vec includes two main models: skip-gram [29] and continuous bag of words (CBOW) [30]. While skip-gram focuses on the probability of generating context words for a central word, CBOW considers the probability of generating the center word from the context words. This paper employs CBOW. FastText [31] proposes an *n*-gram embedding technique where each center word is represented by the sum of the subword vectors around it. Glove [32] is built considering the co-occurrence probabilities with the context.

In previous studies, the most popular method to create a pre-trained embedding model is to treat functions as a paragraph after removing redundant stop-words and symbols [7], [8], [27]. Hence, different statements are placed in the same context. For example, for statements **"int a; b = 1;"**, although no direct relation exists between variables a and b in this case, they are considered close to each other from an NLP perspective. This misunderstanding may inhibit the algorithm's classification ability. Instead of building an embedding model at the function level, we utilize each statement independently in its context to create the pre-trained models.

Node types are added as external information for data encoding. Traversing AST identifies 14 different node types according to their purpose. These include BLOCK, IDENTIFIER, and OPERATOR. We apply one-hot encoding to represent this information. Finally, the encoding is concatenated with textual encoding (via word embedding) as the initial vector representation of a node.

The vector length of the node representation must also be considered. The length should be sufficient for classification and quick computations. Through observations, we found that 86.32% of the sequences on each statement have a length less than or equal to 10 (Fig. 6). To balance the representation of sequences and the machine's processing power, the maximum length of each statement is set to 10 words and symbols. Statements with a length greater than 10 are truncated. Otherwise, a sequence of zeros is appended to the end of the statement. Compared with a previous work [8] in which the word limit for each function was set to 1000, our approach sets the maximum number of nodes to 2000. Thus, the theoretical maximum number of words covered in CAG is up to 20000 words, which is more than enough to cover source code information.

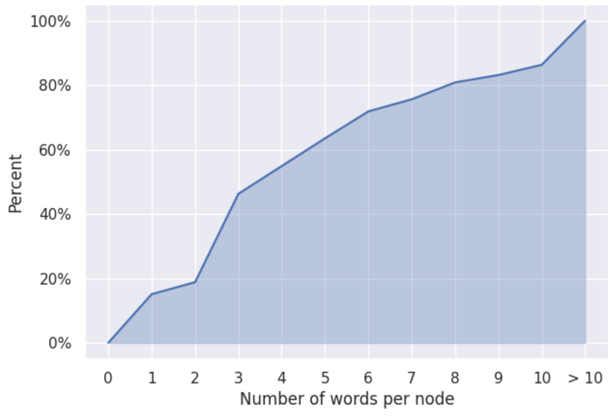


FIGURE 6. Percentage of the number of words per node.

C. TRAINING MODEL

1) GNNs

The results from the data encoding process are used as inputs for the deep learning model. Although convolutional neural networks and recurrent neural networks perform well on matrix and series data, GNN performs well with more structured ones, especially graphs [33]. Hence, GNN is the most suitable model for learning the features for the acquired graph representations.

GNN models can be applied to a wide range of problems as there are many variants. In this paper, we focus on the classification problem for vulnerable code based on a code graph representation. Specifically, three models for graph classification problems are applied: graph convolutional networks (GCNs) [34], gated graph sequence neural networks (GGNNs) [35], and graph isomorphism networks (GINs) [36].

The main idea of GNN is to leverage representation learning. Specifically, a neural network is used to learn input data as graph representations [37]. The graph information such as node features and node connections is used to extract the features of the graph. GNN outputs the embedding for each node. The node embedding contains information on the node itself, its neighboring nodes, and structure information of the entire graph by using core building blocks called message passing layers.

Suppose that $G = (V, E)$ is an input graph where V is the set of nodes and E is the set of edges, and $N(v)$ is the set of neighbor nodes of node $v \in V$. The output of node v after completing the message-passing process is o_v. The message passing process is described as

$$x_{N(v)}^{(t)} = \text{AGGREGATE}^{(t)}(\{h_u^{(t)}, \forall u \in N(v)\}), \quad (1)$$
$$h_v^{(t+1)} = \text{UPDATE}^{(t)}(h_v^{(t)}, x_{N(v)}^{(t)}). \quad (2)$$

At each run t^{th} in the message-passing iteration, $h_v^{(t)}$ and $x_v^{(t)}$ are the hidden embedding and final representation of

TABLE 1. Graph neural networks variants.

Name	Aggregate function	Update function
GCNs	$\sigma(b^{(t)} + \sum_{u \in N(v)} \frac{h_u^{(t)} W^t}{\sqrt{\lvert N(u) \rvert \lvert N(v) \rvert}})$	
GINs	$\sum_{u \in N(v)} e_{vu} h_u^{(t)}$	$MLP((1+\epsilon) h_v^{(t)} + f(h_v^{(t)}))$
GGNNs	$\sum_{u \in N(v)} W_{e_{vu}} h_u^{(t)}$	$GRU(h_v^{(t)}, x_{N(v)}^{(t)})$

MLP: multilayer perceptron GRU: gated recurrent unit

node v, respectively. The function AGGREGATE is responsible for calculating the final representation of node v through the input of all embedding of nodes $N(v)$. The UPDATE function computes the embedding of the next run $h_v^{(t+1)}$ by summing embedding $h_v^{(t)}$ and representation $x_{N(v)}^{(t)}$ of the current run. Note that at time $t = 0$, $h_v^{(0)}$ contains the value of the node features. Finally, after T runs, the value of each node after the message passing is the output of the last layer, which is given by

$$o_v = h_u^{(T)}, \quad \forall u \in N(v). \tag{3}$$

Variants of GNN perform the node embedding computation using the same steps as above. The main difference between the variants is how these models define the AGGREGATE and UPDATE functions. Some commonly used variants of GNN include GCNs, GINs, and GGNNs. Table 1 describes the differences in the implementation of the AGGREGATE and UPDATE functions for select variants. Here, experiments are conducted to provide an overview of the effect of different models on the detection performance.

2) CLASSIFIER

The classifier is a component, which detects vulnerable code based on the input of a graph representation after the message-passing process. This paper classifies vulnerable code based on the graph-level classification. The results are returned after going through the softmax function to compare with the ground truth label

$$y = \sigma\Big(\mathrm{MLP}\big(\Omega(h_v^{(T)}, x_v)\big)\Big), \tag{4}$$

where σ is the softmax function used as the confident index, MLP stands for multilayer perceptron, which is essentially a feedforward network, and Ω is the Global Attention Pooling applied to GNNs, especially in GGNNs. This pooling function [35] can be defined as

$$\Omega = tanh\Big(\sum_{v \in V} \sigma\big(i(h_v^{(T)}, x_v)\big) \otimes tanh\big(j(h_v^{(T)}, x_v)\big)\Big). \tag{5}$$

i and j are neural networks that output vectors from the inputs $h_v^{(T)}$ and x_v. $\sigma(i(h_v^{(T)}, x_v))$ is a soft attention mechanism used to identify nodes that play an important role in graph classification.

V. EXPERIMENTS

A. RESEARCH QUESTIONS

This section experimentally evaluates the performance of CAG for two different datasets. To evaluate the effectiveness of the proposed method, we seek to answer the following research questions:

- RQ1: How does CAG perform on the well-known software assurance reference dataset (SARD) compared to state-of-the-art (SOTA) methods?
- RQ2: Does the proposed code representation in CAG show a better performance than previous SOTA methods for the real-world open-source (RWO) dataset?
- RQ3: How does the performance of CAG compare to other code graph representations?
- RQ4: Do embedding methods affect the result of CAG?
- RQ5: Does the architecture of deep models affect the detection results?
- RQ6: Is CAG suitable for different types of vulnerabilities?

RQ1 and RQ2 aim to validate the effectiveness of the proposed approach in both test-case and real-world source codes. The difference in characteristics of the two datasets provides a fair view to evaluate CAG's performance. RQ1 and RQ2 also aim to establish the baseline performance, which can be compared to SOTA methods. Simultaneously using different code graph representations, word embeddings, and deep neural networks make it difficult to pinpoint where the improvement lies. RQ3–RQ5 aim to identify the factors responsible for improvement by changing one factor while the other two are kept constant. RQ3–RQ5 change code representations, word embeddings, and GNN models, respectively. Finally, RQ6 aims to verify the types of vulnerabilities that CAG is well suited to address.

B. DATASET

Collecting source code data plays a vital role in training the model and evaluating its performance. Multiple methods can be used to create more data such as building synthetic code or collecting actual vulnerability data. The benefit of synthetic code approaches such as SARD [38] is that they can easily manage vulnerabilities. However, the similarities and simplicity between test cases causes models to have a higher performance for synthetic code than for real-world vulnerabilities [8]. Real-world vulnerability data should generate a more diverse data source, leading to a more accurate assessment of the effectiveness of the applied model. With the expansion of open-source software, it is easier to collect a larger range of data sources than those used in previous studies such as MSR [39] and nine-projects dataset [6]. However, differences in the nature of projects such as database construction and coding styles remain a challenge when distinguishing between safe and vulnerable codes.

For these reasons, this study uses both synthetic code (the SARD dataset) and RWO to assess the performance of CAG. RWO is created by gathering real vulnerabilities reported for Common Vulnerabilities and Exposures (CVE) [40] from

TABLE 2. Details of the two vulnerability datasets.

Dataset	#CVE	#CWE	#Vul.	#Non-vul.
SARD dataset	-	-	26,392	23,297
RWO dataset	6,806	99	7,581	120,405

TABLE 3. Confusion matrix.

	Actual positive	Actual negative
Predicted Positive	TP	FP
Predicted Negative	FN	TN

290 different projects. MSR and the nine-projects dataset are used as a reference to enrich the RWO dataset. To ensure that the dataset is current, RWO focuses on recent vulnerabilities and is collected from both Github and NVD [41]. We also checked whether the data is valid and can be converted into a graph. As a result, we collected 6,319 valid vulnerable functions and 120,405 non-vulnerable functions. For the SARD dataset, we also checked the valid state and randomly selected 26,392 valid vulnerable functions and 23,297 non-vulnerable ones. Table 2 shows the details.

C. EXPERIMENTAL SETUP

The RWO dataset is highly imbalanced as the number of negative samples is 16-times higher than the number of positive ones. For more effective training, we randomly extracted 7,600 non-vulnerable functions from the negative samples. Additionally, we used the k-fold cross-validation technique to test the performance of our models and evaluate the overall performance across the entire data set. Here, we divided the data into a training set, test set, and validation set. The validation test set was randomly selected from 20% of the total data. The remaining samples were divided into 5 parts or $k = 5$ in the k-fold cross-validation technique. In each k-fold, we repeated the experiments 20 times for a total of 100 runs. Finally, we calculated the mean of all the runs and used the final result in the evaluation.

To ensure that the differences between CAG and the prior results are statistically significant, we utilized a non-parametric H-test variant for multiple groups. That is, we employed the Kruskal-Wallis test [42]. We chose this test because we ran our models 100 times. Our test is independent of random samples and does not follow a distribution curve. Additionally, the evaluated metrics (accuracy and F1-score) have ordinally scaled characteristics. The null hypothesis is there is no difference between the mean ranks of groups.

D. METRICS

For machine learning problems, it is crucial to precisely evaluate their performance. Although accuracy is often used in classification problems, it does not overview the performance when comparing different algorithms. Therefore, accuracy is used as the main metric to compare methods. Other metrics such as precision, recall, F1-score, and AUC are also employed to ensure an objective evaluation. Since this study uses cross-validation, the metric results are the mean of the different folds. Below, each metric is described.

Precision: The ratio of correctly detected vulnerable samples. Table 3 lists the definitions of the terms TP, FP, TN and FN.

$$Precision(P) = \frac{TP}{TP + FP} \tag{6}$$

Recall: The ratio of correctly detected vulnerable samples relative to the total predicted vulnerable samples.

$$Recall(R) = \frac{TP}{TP + FN} \tag{7}$$

Accuracy: The ratio of correctly detected vulnerable samples to the total samples.

$$Accuracy\ (A) = \frac{TP + TN}{TP + FP + TN + FN} \tag{8}$$

F1-score: The overall performance considering both precision and recall.

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{9}$$

Two popular charts are also employed to clarify the differences between methods: Receiver Characteristic Operator curves (ROC curves) and Precision-Recall curves (PR curves). Both are common metrics used to evaluate binary classification models. The difference is that the ROC curves show the relationship between the true-positive rate and false-positive rate, while the PR curves show the relationship between precision and recall. For imbalanced datasets, PR curves are often preferred. For balanced datasets, the ROC curves and PR curves often show the same trends. The main evaluation metric in this study is the ROC curves because the dataset is balanced. The PR curves are shown as an additional reference. To visually represent the ROC curves as numbers, we used the Area Under Curve of ROC (AUROC). AUROC indicates a model's ability to distinguish classes from each other. AUROC ranges from 0 to 1, where a higher value represents a better discrimination ability. That is, a good model produces a larger AUC value. In this paper, AUC is equivalent to AUROC.

VI. RESULTS

This section shows the experimental results and answers the RQs.

A. RQ1

This first experiment compared the performance of CAG to other SOTA methods using the SARD dataset. CAG was compared to diverse categories of SOTA methods: rule-based approaches (FlawFinder [16] and RATS [15]), similarity-based approaches (VUDDY [13]), recurrent neural networks (BiGRU and BiLSTM [27]), and deep graph network approaches (Devign [28] and BGNN4D [8]). This comprehensive comparison provides an accurate overview of CAG's performance.

TABLE 4. Detection performance of the SOTA approaches and CAG for the SARD dataset.

SARD Dataset					
Approach	A	F1	P	R	AUC
FlawFinder	63.19	57.15	46.21	74.86	-
RATS	57.27	44.55	32.32	71.67	-
VUDDY	80.07	82.20	86.65	78.19	-
BiGRU	96.75	96.99	94.57	99.59	0.992
BiLSTM	96.89	97.14	94.68	**99.72**	0.994
Devign	96.52	96.97	94.65	99.41	0.995
BGNN4D	96.85	97.24	95.26	99.30	0.996
CAG	**97.01**	**97.42**	**95.28**	99.66	**0.996**

TABLE 5. Detection performance of the SOTA approaches and CAG for the RWO dataset.

RWO Dataset					
Approach	A	F1	P	R	AUC
FlawFinder	59.20	37.72	24.75	79.32	-
RATS	55.40	23.42	13.65	**82.21**	-
VUDDY	53.94	61.23	72.83	52.82	-
BiGRU	67.34	58.46	**80.51**	45.59	0.773
BiLSTM	69.87	65.12	77.50	56.15	0.773
Devign	73.83	73.57	74.26	73.13	0.801
BGNN4D	73.63	72.06	76.44	68.21	0.792
CAG	**79.22**	**78.71**	79.75	77.80	**0.857**

Table 4 shows that all machine learning methods outperform rule-based and similarity-based approaches. Rule-based approaches such as RAT and VUDDY showed an accuracy and F1-score around 60% and below 60%, respectively. These values indicate a low performance. Although VUDDY exhibited a better performance with an accuracy and Fl-score around 80%, it is still low. In contrast, all machine learning algorithms achieved an accuracy and F1-score over 96% and 97%, respectively. These results demonstrate the power of deep learning for vulnerability detection.

The results on the SARD dataset were not highly categorical. All the results were around 97%. Although CAG achieved good results, the gap between it and Devign was very small with a difference in accuracy and F1-score of 0.49% and 0.45%, respectively. Hence, it is unclear whether CAG outperforms the other methods. The SARD dataset is a test-case set, which leads to a high performance because these methods can easily classify vulnerable code. The same results also suggest that around 97% is the maximum accuracy achieved using machine learning for the SARD dataset in this experiment. Hereafter, the results using the RWO dataset are analyzed to clarify the differences between the methods.

Conclusion 1: Although CAG works well on the SARD dataset, the results are inconclusive on whether it shows an improvement compared to existing approaches.

B. RQ2

The second experiment repeated the experiment in Section VI-A, except the RWO dataset was used instead of SARD. Because RWO is synthesized from the RWO source code, it should provide a more accurate view of the effectiveness of the different methods in a real environment.

Table 5 summarizes the comparison results. Similar to Section VI-A, this experiment demonstrates the outstanding advantage of machine learning methods. FlawFinder, RATS, and VUDDY gave low accuracies and F1-scores. In the RWO dataset, these approaches never reached more than 60% accuracy. In contrast, the accuracy of the machine learning methods was at least 67%. It is interesting that RATS when performing with this dataset reached up to 82.21% recall, but the low precision score (13.65%) impacted the F1-score.

The correlation of the results between machine learning algorithms on RWO is much clearer than that on SARD. GNN methods outperformed recurrent neural networks on RWO datasets. BiGRU and BiLSTM achieved accuracy below 70%, whereas GNNs had higher accuracy. Deep GNN methods outperformed the other ML approaches. Devign and BGNN4D yielded similar results as they both reached an accuracy and F1-score of 73% and 72%, respectively. CAG gave the highest accuracy (79.22%) and F1-score (78.71%), which are 5.4% and 5.1% higher than Devign, respectively. These results show that CAG not only improves the detection efficiency compared to other methods but also reveals that it is a promising solution for detecting vulnerabilities in real software development.

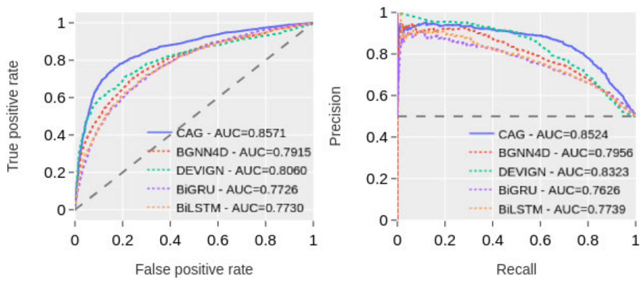


FIGURE 7. ROC curves and PR curves of different SOTA approaches for the RWO dataset.

The ROC curve shown in Fig. 7 verifies the performance of CAG. The detection efficiency increased when using our approach with an AUC value of 0.857. The AUC value was larger than those of BGNN4D (0.779), BiGRU (0.773), and BiLSTM (0.773). Comparing the results with BGNN, the mean accuracy and F1-score of CAG are superior by 5%. This is due to two reasons. First, adding a DT and a forward DT to CAG strengthens deep GNNs' ability to learn the properties of functions. Second, the BGNN algorithm limits the maximum number of nodes to 400 (exceeding this number will exceed the processing limit), which directly limits the ability of deep GNNs, especially when the vulnerable line of code is beyond this range.

Conclusion 2: Using CAG as a code representation improves the performance compared to previous SOTA approaches.

TABLE 6. Comparison between the different code graph representations on vulnerability detection.

RWO Dataset					
Graph	A	F1	P	R	AUC
CDG	72.54	77.44	75.42	79.58	0.786
DDG	75.00	75.91	73.20	78.89	0.830
PDG	75.47	74.54	77.59	71.81	0.820
CFG	76.61	77.51	74.63	80.68	0.835
AST	76.78	78.05	73.27	**83.35**	0.851
CPG	77.11	76.51	77.61	75.60	0.841
CAG	**79.22**	**78.71**	**79.75**	77.80	**0.853**

C. RQ3

The third experiment compared the vulnerability detection performance of CAG to other types of code graph representations. Although the experiments in Section VI-B demonstrated the superior performance of CAG compared to other SOTA approaches, the result is a combination of the network model, word embedding, and graph representation. This experiment analyzed the influence of each component on CAG's performance. Specifically, the experiment compared CAG with six other code graph representations under the same setting conditions. The experiment used GGNN with Word2Vec embedding, and all graphs were run with the same hyperparameters.

Table 6 shows that CAG improves the detection of vulnerable code using a deep learning model. CAG showed the highest accuracy and F1-score. For the RWO dataset, the results are noticeable as CAG outperformed the other graphs in terms of accuracy, F1-score, precision, and AUC. Compared to CPG, CAG shows 2.11% and 2.20% improvements for accuracy and F1-score, respectively. Interestingly, for the recall metric, CAG seemed to be weaker than other graphs such as AST or CFG. However, CAG's F1-score exceeded the other representations because CAG's precision (i.e. ability to select vulnerable code) had a high accuracy.

Figure 8a indicates that CAG may solve the broadest covered area problem. The AUC value of CAG was 0.853 followed by AST (0.851) and CPG (0.841). These results suggest that CAG outperforms super informative graphs such as AST and CPG and is a step toward achieving a robust detection performance of vulnerable code.

Conclusion 3: Among the code graph representations, CAG shows the best performance in terms of accuracy and F1-score.

D. RQ4

The fourth experiment tested the effect of word embedding on identifying vulnerabilities in code functions. Embedding acts as the data kernel of the nodes, allowing GNNs to understand the semantics meaning between statements. As such, the encoding method significantly affects the algorithm's output. However, this factor is often overlooked in previous research. This experiment elucidated the extent that the embedding choice affects the results and whether the dataset influences the choice of embedding method. As described in Section IV-B, we utilized each statement separately to build the pre-trained models, which were re-trained for different datasets.

TABLE 7. Comparison of the effectiveness of the different word embedding methods.

RWO Dataset					
Embedding	A	F1	P	R	AUC
FastText	73.61	74.79	70.95	**79.16**	0.805
Glove	74.92	74.86	74.50	75.46	0.817
Word2Vec	**79.22**	**78.71**	**79.75**	77.80	**0.853**
SARD Dataset					
Embedding	A	F1	P	R	AUC
FastText	**99.32**	**99.39**	99.26	99.53	**0.999**
Glove	96.80	97.20	95.21	99.29	0.996
Word2Vec	97.01	97.42	95.28	**99.66**	0.996

TABLE 8. Comparison of the capability of the different deep graph neural network models.

RWO Dataset					
Model	A	F1	P	R	AUC
GCN	77.99	79.16	74.45	**84.59**	**0.863**
GIN	78.52	**79.20**	76.04	82.80	0.859
GGNN	**79.22**	78.71	**79.75**	77.80	0.853
SARD Dataset					
Model	A	F1	P	R	AUC
GCN	95.20	95.80	93.98	97.70	0.988
GIN	96.99	97.37	**95.41**	99.42	0.995
GGNN	**97.01**	**97.44**	95.30	**99.68**	**0.996**

Table 7 confirms a significant difference between the results of word embedding methods applied to the RWO dataset. Although Word2Vec achieved an accuracy and F1-score of 79.22% and 78.71%, respectively, FastText and Glove achieved around 74% for both metrics. Thus, selecting the proper word embedding significantly increased the accuracy (5%) and F1-score (4%). The ROC curves shown in Fig. 8b suggest that Word2Vec produced superior results compared to other embeddings. Testing on the SARD dataset gave the same results. The accuracy difference between FastText, the best embedding, and the other two methods was 3%, which is noticeable. Consequently, choosing the proper embedding greatly contributes to the improved detection performance.

Table 7 shows that the best embedding differs for the two datasets. While the RWO dataset showed very good results using Word2Vec, FastText was the optimal choice for the SARD dataset. For the SARD dataset, the accuracy when using FastText was up to 99.32% and the F1-score was 99.39%, which are higher than all the results in Section VI-B. Because the appropriate word embedding depends on the dataset, the dataset needs to be analyzed and tested to determine the best embedding.

Conclusion 4: Word embedding greatly affects the detection performance and depends on the dataset.

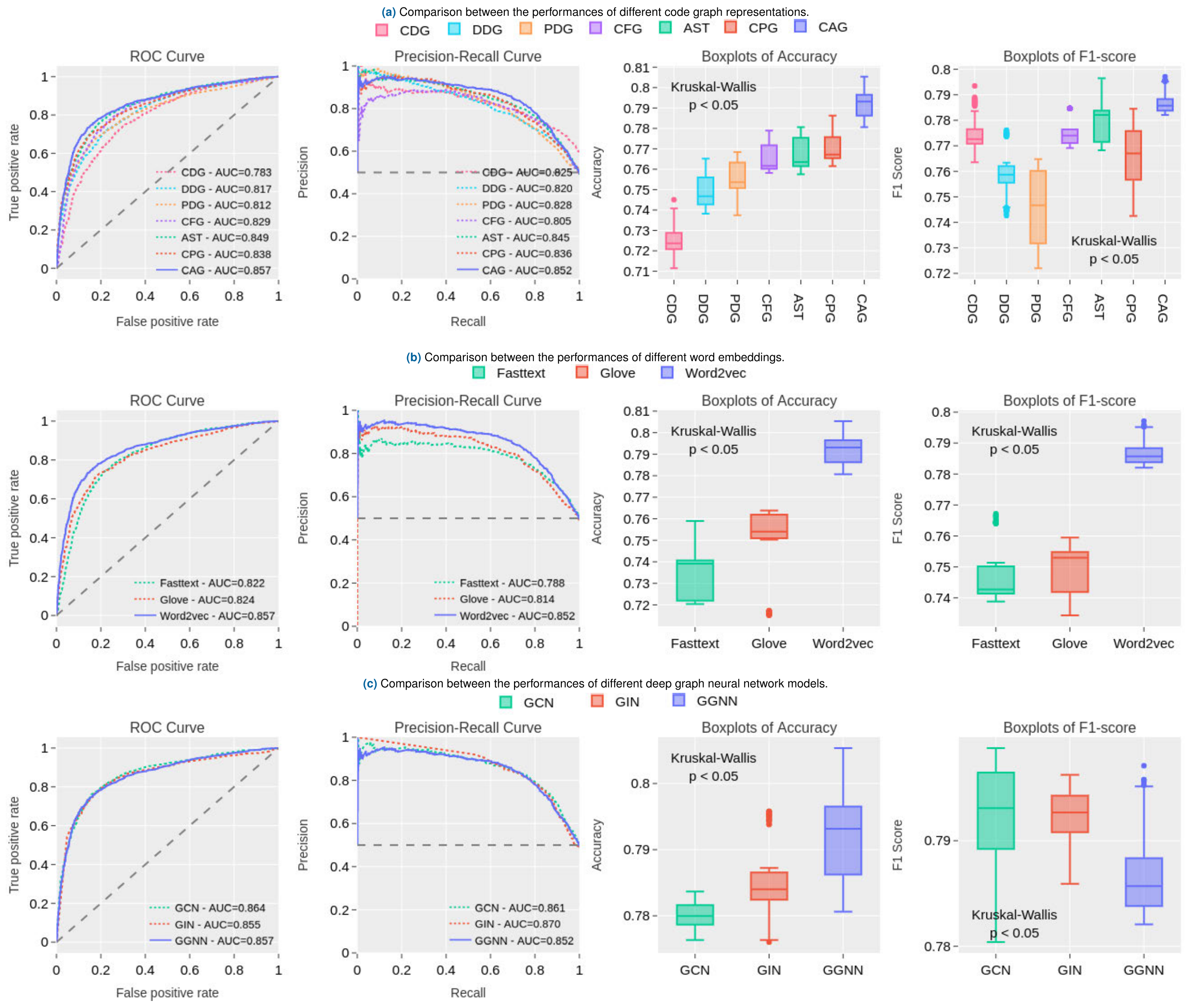


FIGURE 8. ROC curves, Precision-Recall curves, and boxplots of the accuracy and F1-score for different graph representations, word embeddings, and neural network models on the RWO dataset.

E. RQ5

Previous studies did not address the importance of GNNs in vulnerable code detection. The fifth experiment evaluated the influence of different GNNs on detection efficiency. We experimented with deep GNN methods that focus on graph classifications, including GCN, GIN, and GGNN. As mentioned in Section IV-C1, these methods are well-known and directly support the graph classification problem.

Table 8 shows the results for both datasets. GGNN gave slightly better results than the other two algorithms. However, the improvement was insignificant. For the RWO dataset, GGNN gave 1.5% better results than GCN for accuracy. However, GCN produced better results for both the F1-score and recall. The ROC curves for algorithm accuracy were almost identical (Fig. 8c). Additionally, GCN, GIN, and GGNN gave AUC values of 0.864, 0.855, and 0.857, respectively.

The results of the SARD dataset led to a similar conclusion. Although GGNN in this dataset showed a higher performance for both accuracy and F1-score, the differences between GGNN and GIN were insignificant (0.02% for accuracy and 0.07% for F1-score). Although GCN showed the worst performance, the gap between GCN and GGNN for accuracy was less than 2%.

Regardless of the method used, all three models outperformed previous methods and the difference between them is negligible. Thus, the model used to detect vulnerabilities does not play a significant role when the functions are fully semantically and syntactically represented through code graph representations.

TABLE 9. CAG performance on the different types of vulnerabilities.

CWE ID	Description	Recall	Number of samples
CWE-120	Classic Buffer Overflow	90.15	132
CWE-125	Out-of-bounds Read	83.87	564
CWE-400	Uncontrolled Resource Consumption	81.82	132
CWE-284	Improper Access Control	79.61	152
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	78.23	1144
CWE-787	Out-of-bounds Write	76.11	226
CWE-190	Integer Overflow or Wraparound	75.68	259
CWE-20	Improper Input Validation	75.32	555
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	75.08	305
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization	71.84	206

Conclusion 5: The role of the deep neural network on the performance is insignificant. However, choosing the appropriate neural network model can increase the accuracy by 1–2%.

F. RQ6

The final experiment tested the accuracy of CAG to determine whether its performance depended on the type of vulnerability. Table 9 shows the accuracy of the algorithm for different types of vulnerabilities of RWO dataset. The algorithm showed the highest vulnerability accuracy for *"CWE-120 Classic Buffer Overflow"*, and had a recall score up to 90%. The top vulnerabilities correctly identified were closely related to the syntactic structure and the position of the vulnerable code line. This shows that CAG is well suited for detecting vulnerabilities that favor the structure and syntax of the code line. This is reasonable because pattern-based vulnerable detection algorithms tend to be better suited at detecting structural changes rather than semantic ones in functions.

Conclusion 6: The performance of CAG on vulnerabilities varies. CAG is effective for vulnerabilities where it is important to understand the order of code execution.

VII. LIMITATIONS

Our approach contains some limitations. First, our results strongly depended on the datasets utilized. Although both synthetic and real-world code were used to make our assessment as fair as possible, reliability plays an important role in the results. Factors such as labeling and sampling imbalance datasets directly influenced the experiment. Second, this paper focused on applying our method to C/C++ vulnerable code detection, but it should be applicable to other languages. To apply our method to other programming languages, changes are needed to suit the characteristics of the specific programming language. Third, our method only applies to the function level of the source code. Future research should expand the development direction to include file-level code. Fourth, CAG can only detect whether a function is vulnerable but cannot distinguish between different vulnerability types. Fifth, using GNNs creates a high variance problem. Our model overfitted the training set, which caused the results in the test set (shown in this paper) to be lower than expected. In the future, applying regulation solutions should solve this problem.

VIII. CONCLUSION

This study presented code aggregate graphs associated with deep neural networks to increase the efficiency in detecting vulnerabilities in the source code. Specifically, we built CAG to enhance the ability to detect syntactic and semantic source code. CAG is based on the aggregating DT and PDT with other code graph representations. Our method was tested experimentally using two datasets: SARD and RWO datasets. CAG achieved around 80% accuracy and F1-score. These values are higher than those from the previous state-of-the-art algorithms. In addition, the impact of other factors such as word embedding and neural network model on the overall detection performance were evaluated.

In the future, we plan to improve our study in three ways. First, regulation methods are needed to reduce overfitting of the training set and to increase the overall performance because the numerical results indicate that the models had a high variance. Second, we will expand this work to other programming languages. Finally, we will implement an approach that can identify the line of code containing vulnerabilities.

APPENDIX A
NOMENCLATURE

ACRONYMS AND ABBREVIATIONS

GNN	Graph neural networks.
CAG	Code aggregate graph.
DL	Deep learning.
NLP	Natural language processing.
DT	Dominator tree.
PDT	Post-dominator tree.
AST	Abstract syntax tree.
CFG	Control flow graph.
CDG	Control dependence graph.
DDG	Data dependence graph.
PDG	Program dependence graph.
CPG	Code property graph.
GCN	Graph convolutional network.
GIN	Graph isomorphism network.
GGNN	Gated graph convolutional network.

RWO	Real-world open-source.
MLP	Multilayer perception.
GRU	Gated recurrent unit.
SOTA	State-of-the-art.
AUC	Area under curve.

NOTATIONS

G	Graph representation.
V	Set of nodes.
E	Set of edges.
C	Set of source code data.
ζ	Source code data of a node.
γ	Edge type information.
λ	Edge label information.
ϵ	Non-control statements.
$N(v)$	Set of neighbor nodes of v.
$x_v^{(t)}$	Final representation of node v at t^{th} run in message-passing iteration.
$h_v^{(t)}$	Hidden embedding of node v at t^{th} run in message-passing iteration.
o_v	Output of every node after the message-passing process.
σ	Softmax function.
Ω	Global attention pooling.

REFERENCES

[1] White Source Software. (2021). *All About Whitesource's 2021 Open Source Security Vulnerabilities Report*. [Online]. Available: https://www.whitesourcesoftware.com/resources/blog/2021-state-of-open-source-security-vulnerabilities-cheat-sheet

[2] S. Raveendran. (2021). *Explained: How Big is the Damage Caused by Open Source Software Log4J Vulnerability?* [Online]. Available: https://www.onmanorama.com/news/business/2021/12/21/open-source-software-log4j-vulnerability.html

[3] S. Kim, R. Y. C. Kim, and Y. B. Park, "Software vulnerability detection methodology combined with static and dynamic analysis," *Wireless Pers. Commun.*, vol. 89, no. 3, pp. 777–793, Aug. 2016.

[4] G. McGraw, "Software security," *IEEE Secur. Privacy*, vol. 2, no. 2, pp. 80–83, Aug. 2004.

[5] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, "A comparative study of deep learning-based vulnerability detection system," *IEEE Access*, vol. 7, pp. 103184–103197, 2019.

[6] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *Proc. 21st Int. Conf. Inf. Commun. Secur. (ICICS)*, Beijing, China. Berlin, Germany: Springer-Verlag, Dec. 2019, pp. 219–232, doi: 10.1007/978-3-030-41579-2_13.

[7] N. H. Nguyen, V. H. Nguyen, and T. Uehara, "An extended benchmark system of word embedding methods for vulnerability detection," in *Proc. 4th Int. Conf. Future Netw. Distrib. Syst. (ICFNDS)*, 2020, pp. 1–8.

[8] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, Aug. 2021, Art. no. 106576.

[9] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, *arXiv:1801.01681*.

[10] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Depend. Sec. Comput.*, vol. 19, no. 4, pp. 2244–2258, Aug. 2022.

[11] A. Schäfer, W. Amme, and T. S. Heinze, "Detection of similar functions through the use of dominator information," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. Companion (ACSOS-C)*, Aug. 2020, pp. 206–211.

[12] H. Sun, L. Cui, L. Li, Z. Ding, Z. Hao, J. Cui, and P. Liu, "VDSimilar: Vulnerability detection based on code similarity of vulnerabilities and patches," *Comput. Secur.*, vol. 110, Nov. 2021, Art. no. 102417.

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

[14] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a 'Siamese' time delay neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 6, 1993, pp. 1–8.

[15] (2014). *Rough Audit Tool for Security*. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[16] (2018). *Flawfinder*. [Online]. Available: https://dwheeler.com/flawfinder

[17] Y. Younan, W. Joosen, and F. Piessens, "Code injection in C and C++: A survey of vulnerabilities and countermeasures," Dept. Computerwetenschappen, Katholieke Univ. Leuven, Tech. Rep. CW386, Jul. 2004.

[18] Z. Yu, C. Theisen, L. Williams, and T. Menzies, "Improving vulnerability inspection efficiency using active learning," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2401–2420, Nov. 2021.

[19] K. Z. Sultana, "Towards a software vulnerability prediction model using traceable code patterns and software metrics," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2017, pp. 1022–1025.

[20] B. Chernis and R. Verma, "Machine learning methods for software vulnerability detection," in *Proc. 4th ACM Int. Workshop Secur. Privacy Anal.*, 2018, pp. 31–39.

[21] C. A. Welty, "Augmenting abstract syntax trees for program understanding," in *Proc. 12th IEEE Int. Conf. Automated Softw. Eng.*, Nov. 1997, pp. 126–133.

[22] J. Grosch and H. Emmelmann, "A tool box for compiler construction," in *Proc. 3rd Int. Workshop Compiler Compil. (CC)*. Berlin, Germany: Springer-Verlag, 1991, pp. 106–116.

[23] A. Orailoglu and D. D. Gajski, "Flow graph representation," in *Proc. 23rd ACM/IEEE Design Autom. Conf.*, Jul. 1986, pp. 503–509.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604.

[26] G. Ramalingam and T. Reps, "An incremental algorithm for maintaining the dominator tree of a reducible flowgraph," in *Proc. 21st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1994, pp. 287–296.

[27] H. N. Nguyen, S. Teerakanok, A. Inomata, and T. Uehara, "The comparison of word embedding techniques in RNNs for vulnerability detection," in *Proc. ICISSP*, 2021, pp. 109–120.

[28] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[29] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013, pp. 1–9.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.

[31] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. Comput. Linguistics*, vol. 5, pp. 135–146, Dec. 2017.

[32] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1532–1543.

[33] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability detection in C/C++ source code with graph representation learning," in *Proc. IEEE 11th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2021, pp. 1519–1524.

[34] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

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

[36] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018, *arXiv:1810.00826*.

[37] W. L. Hamilton, "Graph representation learning," *Synth. Lect. Artif. Intell. Mach. Learn.*, vol. 14, no. 3, pp. 1–159, 2020.

[38] P. Black, "A software assurance reference dataset: Thousands of programs with known bugs," *J. Res. Nat. Inst. Standards Technol.*, vol. 123, pp. 1–3, Apr. 2018.

[39] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "AC/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020, pp. 508–512.

[40] S. Ozkan. (2011). *CVE Details: The Ultimate Security Vulnerability Datasource*. [Online]. Available: http://www.cvedetails.com

[41] *The National Vulnerability Database (NVD)*. Accessed: Jan. 20, 2022. [Online]. Available: http://nvd.nist.gov

[42] P. E. McKight and J. Najab, ''Kruskal–Wallis test,'' in *The Corsini Encyclopedia of Psychology*. Wiley, 2010, p. 2002.

HOANG VIET NGUYEN received the B.E. degree in computer engineering from the Danang University of Science and Technology, Vietnam, in 2017, and the M.E.Eng. degree in information science and engineering from Ritsumeikan University, Japan, in 2021, where he is currently pursuing the Ph.D. degree with the Cyber Security Laboratory. His current research interests include information security, penetration testing, vulnerability detection, natural language processing, machine learning, and reinforcement learning.

JUNJUN ZHENG (Member, IEEE) received the B.S.E. degree in engineering from Fujian Normal University, Fuzhou, China, in 2010, and the M.S. and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 2013 and 2016, respectively.

From 2016 to 2017, he was a Visiting Researcher at the Department of Information Engineering, Graduate School of Engineering, Hiroshima University. Since 2018, he has been an Assistant Professor with the Department of Information Science and Engineering, Ritsumeikan University, Japan. He is currently an Assistant Professor with the Department of Information Systems Engineering, Graduate School of Information Science and Technology, Osaka University, Japan. His research interests include software reliability, performance evaluation and dependable computing. He is a member of the Operations Research Society of Japan, the Reliability Engineering Association of Japan, the Institute of Electrical, Information and Communication Engineers, and the Institute of Electrical and Electronics Engineers.

ATSUO INOMATA (Member, IEEE) received the D.Sc. degree from the Japan Advanced Institute of Science and Technology, in 2002. He was an Associate Professor at the Information Technology Center, Nara Institute of Science and Technology, in 2016. From 2016 to 2019, he was a Professor at Tokyo Denki University. He has been a Professor with the Cybermedia Center, Osaka University, since 2019. He has also been the Vice-President of Wireless LAN Certification Organization (WiCert) and the Director of Japan Computer Emergency Response Team Coordination Center (JPCERT/CC). His research interests include cryptography and its implementation, risk management, and embedding system security.

TETSUTARO UEHARA (Member, IEEE) received the B.E., M.E., and D.Eng. degrees from Kyoto University, in 1990, 1992, and 1996, respectively. He was an Assistant Professor at the Faculty of Systems Engineering, Wakayama University, from 1996 to 2003. From 2003 to 2005, he was an Associate Professor at the Center for Information Technology, Graduate School of Engineering, Kyoto University. From 2006 to 2011, he was an Associate Professor at the Academic Center for Computing and Media Studies, Kyoto University. From 2011 to 2013, he was the Deputy Director of the Standardization Division in the Ministry of Internal Affairs and Communication, Japan. He has been a Professor with the College of Information Science and Engineering, Ritsumeikan University, since 2013. He has also been the Vice-President of the Institute of Digital Forensics, since 2017. His research interests include systems security, digital forensics, privacy, education in information ethics, and information system management in local government.