The Journal of Systems and Software 214 (2024) 112014

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Code-centric learning-based just-in-time vulnerability detection ☆

Son Nguyen, Thu-Trang Nguyen, Thanh Trong Vu, Thanh-Dat Do, Kien-Tuan Ngo, Hieu Dinh Vo *

Faculty of Information Technology, VNU University of Engineering and Technology, Hanoi, Viet Nam

ARTICLE INFO

Keywords:
Just-in-time vulnerability detection
Code-centric
Code change representation
Graph-based model
Commit-level bugs

ABSTRACT

Attacks against computer systems exploiting software vulnerabilities can cause substantial damage to the cyber infrastructure of our modern society and economy. To minimize the consequences, it is vital to detect and fix vulnerabilities as soon as possible. Just-in-time vulnerability detection (JIT-VD) discovers vulnerability-prone ("dangerous") commits to prevent them from being merged into source code and causing vulnerabilities. By JIT-VD, the commits' authors, who understand the commits properly, can review these dangerous commits and fix them if necessary while the relevant modifications are still fresh in their minds. In this paper, we propose CodeJIT, a novel graph-based code-centric learning-based approach for just-in-time vulnerability detection. The key idea of CodeJIT is that *the meaning of the code changes of a commit is the direct and deciding factor for determining if the commit is dangerous for the code.* Based on that idea, we design a novel graph-based representation, Code Transformation Graph (CTG) to represent the semantics of code changes in terms of both code syntactic structure and program dependencies. A graph neural network (GNN) model is developed to capture the meaning of the code changes represented by our graph-based representation and learn to discriminate between dangerous and safe commits. We conducted experiments to evaluate the JIT-VD performance of CodeJIT on a dataset of 20K+ dangerous and safe commits in 506 real-world projects from 1998 to 2022. Our results show that CodeJIT significantly improves the state-of-the-art JIT-VD methods by up to 66% in Recall, 136% in Precision, and 68% in F1. Moreover, CodeJIT correctly classifies nearly 9/10 of dangerous/safe (benign) commits and even detects 69 commits that fix a vulnerability yet produce other issues in source code.

1. Introduction

Software is a critical element in a broad range of real-world systems. Attacks against computer systems exploiting software vulnerabilities (security defects/bugs), especially in critical systems such as traffic control, aviation coordination, chemical/nuclear industrial operation, and national security systems, can cause substantial damage. The latest survey (Krasner, 2021) in 2021 shows that by 2020, the total cost of bugs related to software vulnerabilities in the US (accounting for 33% of the market share in the whole world technology market) is 1.56 trillion USD. This cost accounts for 75% of the cost caused by poor-quality software. To minimize the consequences caused by vulnerabilities in software, vulnerabilities should be detected and fixed as soon as possible because late detection and correction can cost up to 200 times as much as early correction or even cause more severe damage (Braz et al., 2022; Kang et al., 2021; Lomio et al., 2022). The most popular and effective solution being applied in practice is to use techniques that scan the entire source code and automatically detect the presence of vulnerabilities before the software is released (Lin et al., 2020; Elder et al., 2022).

Most of the existing methods scan all the software components such as files, functions, or lines in source code to detect vulnerabilities (Lin et al., 2020). *Similarity-based* techniques (Kim et al., 2017; Li et al., 2016) can detect vulnerabilities in code that are incurred by code cloning. Meanwhile, *pattern-based* methods (Flawfinder, 2022; SonarQube, 2022; Checkmarx, 2022; Neuhaus et al., 2007) detect vulnerabilities based on the features defined by human experts for representing vulnerabilities, which makes them error-prone and laborious. To impose as little reliance on human experts as possible, researchers recently have proposed many *learning-based* approaches which can effectively detect vulnerabilities caused by a wide range of reasons (Hin et al., 2022; Fu and Tantithamthavorn, 2022; Li et al., 2021a; Ding

☆ Editor: Yan Cai.

* Corresponding author.
E-mail addresses: sonnguyen@vnu.edu.vn (S. Nguyen), trang.nguyen@vnu.edu.vn (T. Nguyen), thanhvu@vnu.edu.vn (T.T. Vu), 20020045@vnu.edu.vn (T. Do), tuanngokien@vnu.edu.vn (K. Ngo), hieuvd@vnu.edu.vn (H.D. Vo).

https://doi.org/10.1016/j.jss.2024.112014
Received 7 August 2023; Received in revised form 5 January 2024; Accepted 27 February 2024
Available online 29 February 2024

et al., 2022b; Li et al., 2018; Zou et al., 2019; Zhou et al., 2019; Duan et al., 2019; Cao et al., 2022; Cheng et al., 2022).

These techniques often report a large number of alarms. After that, the detected suspicious components still need to be manually examined to determine the actual presence of the vulnerability. Such task requires selecting an *appropriate developer group* who can thoroughly understand the business logic of the components. However, this proper developer selection task is also very challenging (Braz et al., 2022; Kang et al., 2021; Lomio et al., 2022). In practice, developers usually commit their changes very regularly. In Mozilla Firefox, the developers recently pushed 10K+ commits to construct a release from the previous version.[1] At the same time, we found that there are some files touched by up to 155 developers in Mozilla Firefox. Thus, with large accumulated sets of commits and authors at the released time, correctly identifying the authors of dangerous code could be non-trivial (Perl et al., 2015; Kang et al., 2021; Braz et al., 2022; Śliwerski et al., 2005a). Even if the authors are correctly identified, the developers might still need to spend much time recalling and drilling down to the files/functions. This could significantly slow down the quality assurance process and development cycle. Thus, although the recommendations based on vulnerability detection at the release time (long-term recommendations) can be useful in some contexts, they have their own drawbacks in practice (Kang et al., 2021; Braz et al., 2022; Lomio et al., 2022).

Meanwhile, *just-in-time* or *commit-level* recommendations (short-term recommendations) could be preferred because they allow commits' authors to get immediate feedback on their code changes. This feedback could help developers faster improve their code quality since the context of the changes is still fresh in their minds. In addition, the just-in-time vulnerability detection techniques facilitate not only the authors in responsibly making code changes but also code auditors in reviewing code commits (Braz et al., 2022; Elder et al., 2022).

The past few years have witnessed a few successful research to investigate *just-in-time vulnerability detection* (aka. *vulnerability detection at the commit-level*) (Yang et al., 2017; Perl et al., 2015) and just-in-time general defect detection (Pornprasit and Tantithamthavorn, 2021; Ni et al., 2022; Zeng et al., 2021; Hoang et al., 2019). These studies utilize commit messages and expert features such as authors' experiences, commit time, and code changes' complexity (*e.g.*, numbers of added/deleted lines of code) to calculate the metrics to determine if commits are suspicious. In fact, these features might have some correlations but not necessarily cause the presence of vulnerabilities at the commit level. For example, a commit whose message contains bug-fixing keywords such as "*fix*", "*failures*", or "*resolves*" could be very likely to be classified as a safe one (Zeng et al., 2021). Although the commit fixes some bugs, the possibility that the commit creates other bugs in the code should not be eliminated. Meanwhile, the studies utilizing the expert features often leverage on certain hypotheses (Perl et al., 2015; Pornprasit and Tantithamthavorn, 2021; Ni et al., 2022; Chakraborty et al., 2021; Zeng et al., 2021) such as the larger number of added lines of code, the higher possibility the commits contain vulnerabilities (Zeng et al., 2021). Consequently, the dangerous commits with a few added lines of code could be misidentified. Thus, commit messages and the expert features could correlate but not necessarily result in the presence of vulnerabilities, which are, in fact, brought about by code changes.

In this paper, we introduce CodeJIT, a novel ***code-centric*** learning-based approach which focuses on capturing the meaning of code changes in known dangerous or safe (benign) commits, rather than utilizing commit messages or the expert features, to detect vulnerabilities at the commit level. Our idea is that *for a commit, the meaning of the changes in the source code is the direct and deciding factor for assessing the commit's suspiciousness.* This idea is reasonable because a commit should be considered dangerous if its code changes carry vulnerabilities or interact with the unchanged code to collectively introduce vulnerabilities once the changed code is merged into the source code.

To implement our code-centric idea for JIT-VD, we propose a novel graph-based code transformation representation (**C**ode **T**ransformation **G**raph - CTG) to capture the semantics of the code changes caused by commits by representing the changed code in relation to the related unchanged code. Particularly, the CTG of a commit contains rich information about the code changes/transformations in both the syntactic structure and program dependencies, which are necessary to capture the syntax and semantics of the code changes. To construct the CTG of a commit, the code versions before and after the commit are analyzed to build the graphs representing the syntactic structure and program dependencies before and after the commit. Based on the resulting graphs of the versions before and after the commit, the CTG is constructed from the set of nodes which are *added*, *deleted*, and *unchanged* code elements such as statements, expressions, or operators. The CTG's nodes are connected by the edges, which are the *added*, *deleted*, and *unchanged* syntactic structure and dependency relations between the nodes.

Next, CodeJIT relies on the code-related aspects of commits, which are explicitly represented by their CTGs, to learn discriminating dangerous/safe commits. Particularly, CTGs can be intuitively treated as relational graphs. A Relational Graph Convolutional Network (RGCN) model is developed to capture the knowledge from the CTGs of the prior known safe/dangerous commits and learn to detect dangerous vulnerabilities at the commit level.

We conducted experiments to evaluate the JIT-VD performance of CodeJIT on a large dataset of 20K+ dangerous/safe commits in 506 C/C++ projects from 1998 to 2022. We compared CodeJIT with the state-of-the-art JIT-VD approaches (Perl et al., 2015; Hoang et al., 2019; Pornprasit and Tantithamthavorn, 2021; Zeng et al., 2021; Ni et al., 2022). Our results show that CodeJIT significantly improves the state-of-the-art JIT-VD techniques by up to 68% in F1 and 77% in classification accuracy. Especially, CodeJIT can correctly distinguish nearly 9/10 of dangerous/safe commits. The recall of CodeJIT is up to 66% better than that of the existing approaches. Moreover, CodeJIT can achieve the precision of 90%, which is 32%–123% better than the precision of the other approaches. In other words, 9/10 suspicious commits detected by CodeJIT actually are dangerous. Hence, with CodeJIT, practitioners could discover more the commits dangerous for their code and spend less undesirable efforts for the false alarms during the security inspection process.

In brief, this paper makes the following contributions:

1. Code Transformation Graph: A novel graph-based representation of code changes to capture the changes in the crucial aspects of code: the syntactic structure and program dependencies.
2. CodeJIT: A novel graph-based code-centric approach for detecting vulnerabilities at the commit level.
3. An extensive experimental evaluation showing the performance of CodeJIT over the state-of-the-art methods for just-in-time vulnerability detection.
4. A public dataset of 20K+ safe and dangerous commits collected from 506 real-world projects, which can be used as a benchmark for evaluating related work.

The detailed implementation of CodeJIT and dataset can be found at: https://github.com/ttrangnguyen/CodeJIT.

2. Motivating example

We first illustrate the problem of just-in-time vulnerability detection and explain the motivation for our solution.

[1] From `release_v109` to `release_v110` more details: https://github.com/mozilla-mobile/fenix.

```
460 461    if (ff_alloc_extradata(st->codec, len))
461 462        return AVERROR(ENOMEM);
462        - avio_read(pb, st->codec->extradata, len);
    463    + if ((ret = avio_read(pb, st->codec->extradata, len)) != len)
    464    +     return ret < 0 ? ret : AVERROR_INVALIDDATA;
```

(a) Commit `2c635fa` fixing a vulnerability yet causing another

```
461 461    if (ff_alloc_extradata(st->codec, len))
462 462        return AVERROR(ENOMEM);
463 463    if ((ret = avio_read(pb, st->codec->extradata, len)) != len) {
    464    +     av_freep(&st->codec->extradata);
    465    +     st->codec->extradata_size = 0;
464 466        return ret < 0 ? ret : AVERROR_INVALIDDATA;
```

(b) Commit `ac480cb` fixing the vulnerability in Figure 1a

Fig. 1. Two commits in *FFmpeg* project demonstrating the importance of understanding code changes for JIT-VD.

2.1. Guiding insight

Guiding insight 1. *For a commit, the code changes' meaning is the reliable factor for deciding if the commit is dangerous or not*. Intuitively, a commit is dangerous to source code if it contains dangerous code changes either having vulnerabilities or introducing the vulnerabilities caused by the interaction between the changed code and the unchanged code. For example, the commit in Fig. 1(a) contains vulnerable changed code itself, while the commit in Fig. 2(a) brings a vulnerability to source code due to the relation between the changed and the unchanged code. Specifically, in Fig. 1(a), the added code itself contains a memory leak issue. The commit replaces the function call `avio_read()` (at line 462, the old version) by a `if`-block examining the returned value of the call (lines 463–464, the new version). In the new version, `avio_read()` allocates the memory of `st->codec->extradata`, yet the author forgot to free `extradata` when the values of `ret` and `len` are different (line 463 in the new version). In Fig. 2(a), adding `av_free_packet(&pkt)` (line 1352, the new version) to free the packet data allocated by calling `av_read_frame` at line 1346 only when decoding audio frames. Consequently, the added code and the existing code collectively cause a leak problem.

Besides the code changes, to determine if a commit is suspiciously dangerous, ones could rely on expert features and commit messages (Zeng et al., 2021; Perl et al., 2015; Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021). Commit messages and expert features such as commit authors' experience or the number of added lines might have some relations to the suspiciousness of commits. However, they might be not necessarily discriminative for detecting dangerous commits. For example, safe commits and dangerous commits could have the same number of added lines, and be produced by the same author (*e.g.*, the dangerous commit in Fig. 1(a) and the safe one in Fig. 1(b)). The application of commit messages and the expert features for JIT-VD is discussed in Section 6.

Guiding insight 2. *To precisely capture the semantics of code changes, besides the changed parts, the unchanged code is also necessary*. Indeed, the unchanged code could connect the related changed parts and help to understand the code changes as whole. In Fig. 3, without considering the unchanged parts (*e.g.*, line 47, the old version/line 48, the new version), the changed parts (lines 25 and 49, the new version) can be considered semantically irrelevant. However, the added lines 24–25 affect the behavior of the added line 49 via the unchanged statement (line 48, the new one).

Additionally, once changed statements are introduced to a program, they change the program's behaviors by interacting with the unchanged statements via certain code relations such as control/data dependencies. Hence, precisely distinguishing similar changed parts could require their relations with the unchanged parts. For example, although the atomic changes adding `av_free_packet(&pkt)` in Figs. 2(a) and 2(b) are apparently similar, they are semantically different in their related unchanged parts. Adding `av_free_packet(&pkt)` in Fig. 2(a) is to free the packet data only when decoding audio frames, which causes a leak problem after `7328c2f`. Meanwhile, the adding one in Fig. 2(b) (line 1353, the new version) at each demuxing loop iteration, not only when decoding. This safe commit fixes the unintentional problem caused by the dangerous commit `7328c2f` in Fig. 2(a). The suspiciousness of commit `7328c2f` (Fig. 2(a)) and commit `4fd1e2e` (Fig. 2(b)) could not be determined without considering the relation between the changed code and the unchanged code. Thus, *not only the changed parts but also the related unchanged parts are necessary for representing code changes to precisely capture their meaning and determine their suspiciousness*.

```
1346 1346    while (!av_read_frame(fmt_ctx, &pkt)) {
  ...          //...
1349 1349      if (do_show_frames &&
1350 1350          get_video_frame(fmt_ctx, &frame, &pkt)) {
1351 1351          show_frame(w, &frame, fmt_ctx->streams[pkt ...;
1352       -      av_destruct_packet(&pkt);
     1352   +      av_free_packet(&pkt);
1353 1353      }
```

(a) Commit `7328c2f` causing a vulnerability in *FFmpeg*

```
1346 1346    while (!av_read_frame(fmt_ctx, &pkt)) {
  ...          //...
1349 1349      if (do_show_frames &&
1350 1350          get_video_frame(fmt_ctx, &frame, &pkt)) {
1351 1351          show_frame(w, &frame, fmt_ctx->streams[pkt ...;
1352       -      av_free_packet(&pkt);
1353 1352      }
     1353   + av_free_packet(&pkt);
```

(b) Commit `4fd1e2e` fixing the issue in Figure 2a

Fig. 2. Two commits in *FFmpeg* project demonstrating the importance of considering both changed and unchanged code in understanding code changes.

```
       24   +    if (update_auto_var)
       25   +        update_var = 0;
47     48        if (update_var) {
48        -        set_band_parameters(s, dnch);
       49   +        set_band_parameters(dnch, s);
```

Fig. 3. Unchanged code connects changed parts.

Guiding insight 3. *The changes in both the syntactic structure and program dependencies are necessary and provide rich information for understanding and determining the suspiciousness of code changes*. For example, in Fig. 2(b), the memory leaking issue is fixed by the changes in the dependencies of the function called `av_free_packet`. Indeed, changing the call of `av_free_packet` from line 1352 to line 1353 (outside the `if` block) causes the call no longer control-dependent on the `if`-statement (line 1349), and creates a new control dependency of the call on the `while`-statement at line 1346. As a result, instead of only when decoding (inside the `if` block), `av_free_packet` is always called in the `while`-loop to free the packet data. Thus, it is important to capture the dependency relation of the code changes.

Additionally, the *Buffer Overflow* issue shown in Fig. 4 is brought by the changes in the syntactic structure. Specifically, while the dependencies between the statements remain unchanged, the right-hand side of the comparison at line 4 is changed from `BUF_SIZE` to `2*BUF_SIZE`. This allows line 5 to copy the amount of data larger than the capacity

of `buf`. This demonstrates the importance of the syntactic structure relation in understanding the code changes.

Overall, these guiding insights indicate that the code changes of a commit are the *deciding factor* in determining the suspiciousness of the commit. To *precisely* capture the changes' meaning, not only the changed code but also the related unchanged parts are required. The changes in the code syntactic structure and program dependencies are essential to *comprehensively* represent the meaning of the changes.

2.2. Key ideas

Based on the guiding insights, we introduce CodeJIT centralizing the role of code changes in JIT-VD and considering the changed code of commits in relation to the unchanged code to assess commits' suspiciousness. For CodeJIT to work, we rely on the following key ideas:

1. We develop a graph-based representation for the code changes of commits, namely *Code Transformation Graph (CTG)*. The representation captures the changes/transformations in both the code syntactic structure and program dependencies of the changed code and the related unchanged code.
2. A relational graph neural network model is employed to capture the knowledge of the known dangerous/safe commits represented in CTGs and evaluate the suspiciousness of newly encountering commits.

3. Semantic code change representation

In literature, several techniques have been proposed to represent code changes in flat sequences (Liu et al., 2020; Nie et al., 2021; Wang et al., 2021; Wu et al., 2022). However, a few graph-based change representations have been developed to capture the relation between the code versions before and after the changes, such as CPath-Miner (Nguyen et al., 2019) and FIRA (Dong et al., 2022). In CPath-Miner (Nguyen et al., 2019), the code versions before and after a commit are represented as two program dependence graphs (PDGs). These PDGs are mapped to form a graph representing the code change. Meanwhile, FIRA (Dong et al., 2022) maps the two abstract syntax trees (ASTs) of the code version before and after the commit to represent the code change. However, none of them dedicate to represent code changes in both syntactic structure and program dependencies, which are both essential for JIT-VD. In this work, we introduce our novel graph-based code change representation which explicitly expresses the transformation in both syntactic structure and program dependencies. Particularly, the code versions before and after a commit are represented by *Relational Code Graphs* (Section 3.1), which capture both syntactic structure and program dependencies in the code. After that, these graphs are used to construct the *Code Transformation Graph (CTG)* (Section 3.2) to capture the changes caused by the corresponding commit. The CTG is reduced to eliminate the irrelevance of the changes and retain the essential elements in the graph.

3.1. Relational code graph

From our point of view, a program is a set of *inter-related elements* and *relations* with others. Essentially, both the syntactic and semantic aspects of programs are necessary to properly understand the program (Yamaguchi et al., 2014; Ngo et al., 2021). In this work, we use a graph-based representation, *Relational Code Graph*, for programs that represents the code elements as well as the syntactic aspect via the relation of *syntactic structure* (Yamaguchi et al., 2014) and the semantic aspect via the relation of *program dependency* (Ferrante et al., 1987). Particularly, the relation of syntactic structure in code reflects how code elements in the code are organized following the syntactic rules of the programming language at hand. Meanwhile, the relation of program dependency expresses the data/control dependencies between code statements/elements in the code snippet.

```
1  1    void str_cpy(char *str, size_t n) {
2  2        char buf[BUF_SIZE], *ar;
3  3        size_t len = strlen(str);
4     -     if(len < BUF_SIZE) {
      4  +  if(len < 2*BUF_SIZE) {
5  5            memcpy(buf, str, len);
6  6        }
7  7    }
```

Fig. 4. A dangerous commit causing a *Buffer Overflow* vulnerability.

Definition 1 (*Relational Code Graph*)**.** For a code snippet f, the corresponding relational code graph, a directed graph $G_f = \langle N, E, \mathcal{R} \rangle$, represents the syntactic structure and program dependencies of f. Formally, $G_f = \langle N, E, \mathcal{R} \rangle$ is defined as followings:

- N is a set of nodes which are the AST nodes of f. In N, the leaf nodes are code tokens, while internal (non-leaf) nodes are abstract syntactic code elements (*e.g.*, statements, predicate expressions, assignments, or function calls).
- E is a set of edges that represent certain relations between nodes, such relations are either structure or dependency. For $n_i, n_j \in N$, an edge exists from node n_i to node n_j regarding relation $r \in \mathcal{R} = \{syntactic\ structure, program\ dependency\}$, $\exists e^r_{ij} = \langle n_i, r, n_j \rangle \in E$ if there is a relation r between node n_i and node n_j.

Note that the *program dependency* edges connect the nodes, which are statements or predicate expressions (Ferrante et al., 1987). Each statement/predicate expression contains the descendants connected via the *syntactic structure* edges. The reason for the incorporation of multiple kinds of knowledge about code in a single representation instead of using multiple graphs is that syntactical information could explain the dependencies between nodes. For example, Fig. 5 shows the relational code graphs of the function before and after the change in Fig. 4. Statement s at line 3 of Fig. 4 assigns (=) a value to variable `len` which is used by statement s' at line 4. There is a data-dependency relation from the statement node corresponding to s to the statement node corresponding to s' (Fig. 5). Explicitly representing the structures of s (variable `len`, assignment operator, and `strlen(str)`) and s' (`len` as an argument of `memcpy`) could help explain the existence of the dependency edge from s to s'. In this work, we use Joern analyzer (Yamaguchi et al., 2014) to analyze code and determine the structure and dependency relations between code elements to construct relational code graphs.

3.2. Code transformation graph

In this work, we aim to represent the code transformations by commits in both the syntactic structure and the dependencies using graphs. As explained in Section 2, the changes should be represented in the context of the unchanged parts, as well as the transformations in the code elements and the relations between the elements.

Definition 2 (*Code Transformation Graph (CTG)*)**.** For a commit changing code from a version to another, the *code transformation graph* is an annotated graph constructed from the relational code graphs of these two versions. Formally, for $G_o = \langle N_o, E_o, \mathcal{R} \rangle$ and $G_n = \langle N_n, E_n, \mathcal{R} \rangle$ which are the relational code graphs of the old version and the new version respectively, the CTG $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}, \mathcal{R}, \alpha \rangle$ is defined as followings:

- $\mathcal{N}$ consists of the code elements in the changed and unchanged parts, $\mathcal{N} = N_o \cup N_n$.
- $\mathcal{E}$ is the set of the edges representing the relations between nodes, $\mathcal{E} = E_o \cup E_n$.
- $\mathcal{R}$ is a set of the considered relations between code elements, $\mathcal{R} = \{syntactic\ structure, program\ dependency1tt$.

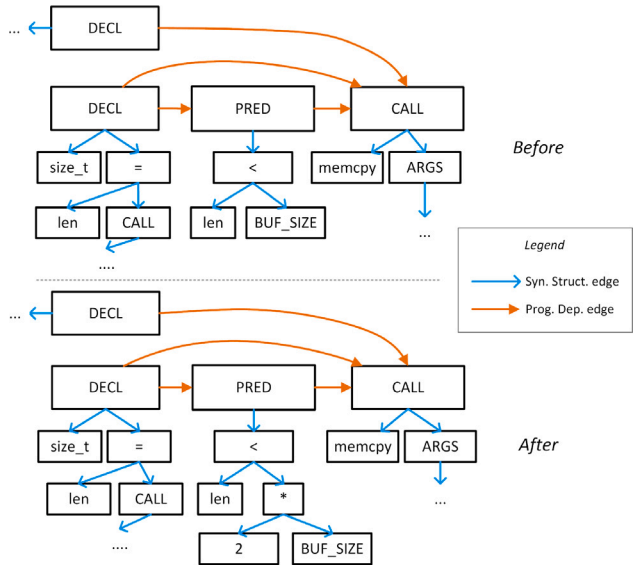

Fig. 5. The relational code graphs before and after the change in Fig. 4.

- Annotations for nodes and edges are either *unchanged*, *added*, or *deleted* by the change. Formally, $\alpha(g) \in \{unchanged, added, deleted\}$, where g is a node in $\mathcal{N}$ or an edge in $\mathcal{E}$:
  - $\alpha(g) = added$ if g is a node and $g \in N_n \setminus N_o$, or g is an edge and $g \in E_n \setminus E_o$
  - $\alpha(g) = deleted$ if g is a node and $g \in N_o \setminus N_n$, or g is an edge and $g \in E_o \setminus E_n$
  - Otherwise, $\alpha(g) = unchanged$

The CTG of the change in Fig. 4 is partially shown in Fig. 6. As we could see the *syntactic structure* edges, the structure of the predicate expression at line 4 is changed. Specifically, `2*BUF_SIZE` is replaced by `BUF_SIZE` in the right-hand side of the less-than comparison expression. This allows the statement at line 5 to be executed even when `len > BUF_SIZE`. In the CTG, through PDG edges, the statement copies `len` (defined at line 3) bytes from `str` (declared at line 1) to `buf` (declared at line 2) which can have maximum `BUF_SIZE` bytes or even greater. This will cause an overflow error when `2*BUF_SIZE > len > BUF_SIZE`.

Theoretically, a CTG corresponding to a commit could be very large and contain many unchanged nodes/edges which are irrelevant to the changes. These irrelevant parts might be unnecessary to capture the changes' meaning and also produce noise for understanding the changes. To trim those irrelevant parts, the nodes/edges that are not relevant to the changed nodes/edges are removed from $\mathcal{G}$. Our idea is that all the statements/predicate expressions containing changed nodes and their related statements/expressions via dependencies are kept. Additionally, the descendant nodes of these statements/expressions are also retained in the graph to enable further elaboration. Formally, given CTG $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}, \mathcal{R}, \alpha \rangle$, a node n is considered as *relevant* to the change if n satisfies one of the following conditions:

- n is a statement/predicate expression node which is the ancestor of a deleted/added node n'. In other words, there is a path from n to n' only via *syntactic structure* -edges.
- n is a statement/predicate expression node, and n impact/is impacted by a *relevant* node n'. In other words, there is a path from n to n' or from n' to n via *program dependency* - edges.
- n is a descendant of a relevant node n'. In other words, there is a path from n' to n only via *syntactic structure* - edges.

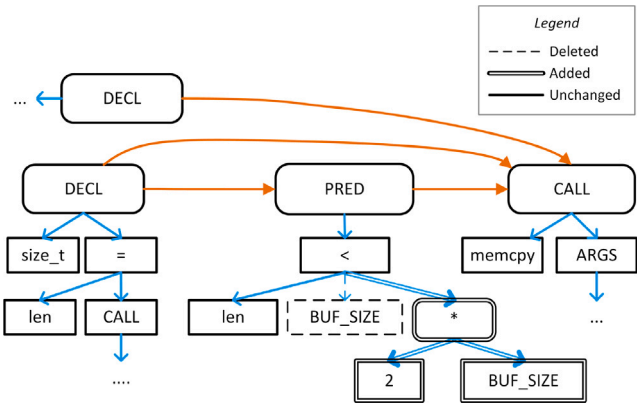

Fig. 6. The CTG corresponding to the change in Fig. 4.

After trimming, the simplified CTG contains the nodes which are relevant to the changes. The edges which do not connect any pair of nodes are also removed from the graph.

4. Code-centric just-in-time vulnerability detection with graph neural networks

Fig. 7 shows the overview of our JIT-VD model. In this work, each CTG is treated as a relational graph. To capture the important features of the graphs, not only the contents of the nodes but also their relations need to be considered in JIT-VD. For this purpose, we apply relational graph neural networks (Busbridge et al., 2019; Schlichtkrull et al., 2018) for the JIT-VD task.

First, the nodes are embedded into d-dimensional hidden features n_i produced by embedding the content of the nodes. Particularly, to build the vectors for nodes' content, we use Word2vec (Mikolov et al., 2013) which is widely used to capture semantic similarity among code tokens (Ding et al., 2022a). Then, to form the node feature vectors, the node embedding vectors are annotated with the change operators (*added*, *deleted*, and *unchanged*) by concatenating corresponding one-hot vector of the operators to the embedded vectors, $h_i^0 = n_i \bigoplus \alpha(n_i)$, where $\bigoplus$ is the concatenating operator and α returns the one-hot vector corresponding the annotation of node i. The resulting vectors are fed to a relational graph neural network model which could be a relational graph convolution network (RGCN) (Schlichtkrull et al., 2018) or a relational graph attention (RGAT) model (Busbridge et al., 2019). Each layer of RGCN or RGAT computes the representations for the nodes of the graph through message passing, where each node gathers features from its neighbors under every relation to represent the local graph structure. Stacking L layers allows the network to build node representations from the L-hop neighborhood of each node. Thus, the feature vector h_i^{l+1} at node i at the next layer is:

$$h_i^{l+1} = \sigma\left(\sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_i^r} \frac{1}{c_{i,r}} g_j^r\right) \quad (1)$$

where $g_j = W_r^l h_j^l$ is the distinct intermediate representation of node j under relation r, W_r^l is a learnable weight matrix for feature transformation specific for relation r. In Eq. (1), $\mathcal{N}_i^r$ is the set of neighbor indices of node i under relation $r \in \mathcal{R} = \{structure, dependency\}$. Additionally, $1/c_i^r$ is a problem-specific normalization constant that can be chosen in advance (such as $c_i^r = |\mathcal{N}_i^r|$). Meanwhile, σ is a non-linear activation function such as ReLU. In this work, instead of using constant $1/c_i^r = |\mathcal{N}_i^r|^{-1}$, we replace the normalization constant by an learnable attention weight a_{ij}^r, where $\sum_{j,r} a_{ij}^r = 1$, to allow the model learn the

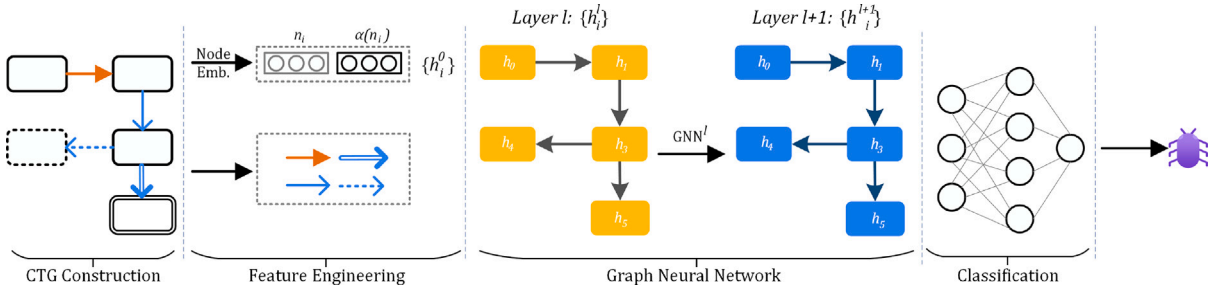

Fig. 7. Just-In-Time Vulnerability Detection Model in CodeJIT.

importance of node j and the relation between node i and node j under relation r (Busbridge et al., 2019):

$$a_{ij}^r = \text{softmax}_j(E_{ij}^r) = \frac{\exp(E_{ij}^r)}{\sum_{k\in\mathcal{N}_i^r}\exp(E_{ik}^r)}$$

$$E_{ij}^r = \text{LeakyReLu}(q_i^r + k_j^r)$$

$$q_i^r = g_i^r Q^r, \text{ and } k_j^r = g_j^r K^r$$

where Q^r and K^r are the query kernel and key kernel projecting g_i (and g_j) into query and key representations (Busbridge et al., 2019). Q^r and K^r are combined to form the attention kernel for relation r, $A^r = Q^r \bigoplus K^r$.

After L GNN layers, a d-dimensional graph-level vector representation H for the whole CTG $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}, \alpha \rangle$ is built by aggregating over all node features in the final GNN layer, $H = \Phi_{i\in[1,|\mathcal{N}|]} h_i^L$, where Φ is a graph readout function such as *sum*, *average*, or *max*. The impact of the aggregation function Φ on CodeJIT's performance will be empirically shown in Section 6. Finally, the graph features are then passed to a Multilayer perceptron to classify if $\mathcal{G}$ is dangerous or not.

5. Evaluation methodology

To evaluate our just-in-time vulnerability detection approach, we seek to answer the following research questions:

RQ1: *Accuracy and Comparison*. How accurate is CodeJIT in detecting dangerous commits? And how is it compared to the state-of-the-art approaches (Perl et al., 2015; Hoang et al., 2020; Li et al., 2022)?

RQ2: *Intrinsic Analysis*. How do the JIT-VD model's properties/components, including the GNN model, the number of GNN layers, and the graph readout function in CodeJIT impact CodeJIT's performance?

RQ3: *Change Representation Analysis*. How do related unchanged parts in CTGs impact CodeJIT's performance? And, how do the structure relation and dependency relation in CTGs impact CodeJIT's performance?

RQ4: *Sensitivity Analysis*. How do various the input's factors, including training data size and changed code's complexity affect CodeJIT's performance?

RQ5: *Time Complexity*. What is CodeJIT's running time?

5.1. Dataset

Recently, Ni et al. constructed a dataset JIT-Defects4J (Ni et al., 2022), which contains 2,332 general just-in-time bugs in Java (*i.e.*, general commit-level bugs), while our specifically targets just-in-time vulnerabilities (*i.e.*, commit-level vulnerabilities). To facilitate applying advanced ML techniques to automatically and effectively learn latent and abstract dangerous/vulnerable commit patterns, we collected a large number of dangerous and safe commits in numerous real-world projects based on the well-known SZZ algorithm (Śliwerski et al., 2005b). We also improved the process of collecting dangerous and safe commits in the existing work (Perl et al., 2015; Yang et al., 2017; Lomio et al., 2022).

Particularly, to identify *dangerous commits/vulnerability contributing commits (VCCs)* (Perl et al., 2015), the general idea is to start from vulnerable statements and identify the previous commits which last modify these statements.[2] Specifically, the vulnerable statements are identified from the *vulnerability-fixing* commits based on the following heuristics. The statements which are deleted by the fixing commits could be considered to be vulnerable (Perl et al., 2015; Yang et al., 2017; Iannone et al., 2022; Fan et al., 2020). For the statements added by the fixing commits, the existing studies (Perl et al., 2015; Yang et al., 2017) assume that such statements are added to fix the nearby statements. In particular, the statements which are *physically* surrounding the added statements are considered to be vulnerable regardless of whether these statements semantically relate to added statements or not. As a result, this procedure could miss the actual VCCs and instead blame commits which are irrelevant. To reduce this risk, we blame the statements which are *semantically* related to the added statements via data/control dependencies rather than the physically surrounding ones.

In practice, forming a vulnerability in a code version could require one or more VCCs over time. However, the existing studies (Lomio et al., 2022; Fan et al., 2019) illustrate that various of VCCs are false positives due to refactoring operators or tangling problems of the fixing commits. For collecting VCCs, although multiple improvements of SZZ have been proposed, this is still an inevitable issue (Fan et al., 2019). Moreover, warning developers by providing any potential VCCs could frustrate them and make the developers spend undesirable efforts. Thus, in this work, to reduce noises while still maintaining the purpose of detecting vulnerabilities as soon as before it is merged to source code, we consider *vulnerability triggering commits* as *dangerous* ones which are the last VCCs triggering to fully expose vulnerabilities. For a commit irrelevant to a known vulnerability, it might still contain unknown vulnerabilities (Perl et al., 2015; Chakraborty et al., 2021; Yang et al., 2017). Meanwhile, the fixing commits had already fixed some problems, been reviewed, and reported to the community. Thus, it could be reasonable to consider the fixing commits, which are not a VCC, as *safe commits*.

In this work, we extract the vulnerability-fixing commits from various public vulnerability datasets (Bhandari et al., 2021; Fan et al., 2020; Zhou et al., 2019). In total, we collected 20,274 commits, including 11,299 safe commits and 8975 dangerous commits from the vulnerabilities reported from Aug 1998-Aug 2022 in real-world 506 C/C++ projects such as FFmpeg, Qemu, Linux, and Tensorflow. Table 1 shows the overview of our dataset. The details of our dataset can be found at https://github.com/ttrangnguyen/CodeJIT.

5.2. Evaluation setup, procedure, and metrics

5.2.1. Empirical procedure

RQ1. Accuracy and Comparison.

Baselines. We compared CodeJIT against the state-of-the-art JIT-VD approaches. As vulnerabilities are security bugs, we also adapted

[2] This can be done by using `git blame`.

Table 1
Dataset statistics.

	#Safe commits	#Dangerous commits	%adds	$\|N\|/\|E\|$ of CTGs
Ffmpeg	4,449	3,462	73.95	0.70
Qemu	3,551	3,183	76.80	0.68
Linux	783	780	78.45	0.66
Tensorflow	224	189	81.48	1.18
		502 projects more...		
Total	11,299	8,975	73.31	0.70

Table 2
Two evaluation settings: *dev.-process* and *cross-project*.

Setting		#Dangerous commits	#Safe commits	#Commits
Dev.-process	Train	7,748	8,471	16,219
	Test	1,227	2,828	4,055
Cross-project	Train	7,714	9,176	16,890
	Test	1,261	2,123	3,384

the advanced approaches for just-in-time bug detection for JIT-VD and compared with CodeJIT:

(1) **VCCFinder** (Perl et al., 2015), which uses commit messages and expert features to train a Support Vector Machine and is specialized for the JIT-VD task.

(2) **CC2Vec** (Hoang et al., 2020) + **DeepJIT** (Hoang et al., 2019): A CNN-based just-in-time defect prediction that the features of commits are captured by CC2Vec, a pre-trained model vectorizing commits using their changed code and commit message;

(3) **JITLine** (Pornprasit and Tantithamthavorn, 2021): A simple but effective method utilizing changed code and expert features to detect buggy commits;

(4) **LA_Predict** (Zeng et al., 2021): A regression-based approach simply using the added-line-number feature which can outperform CC2Vec and DeepJIT; and

(5) **JITFine** (Ni et al., 2022): A DL-based approach extracting features of commits from changed code and commit message using CodeBERT as well as expert features.

For all the baseline approaches, we used the implementation in their original papers.

Procedure. In this comparative study, we evaluate the performance of the approaches in two real-world settings used in the existing studies (Zeng et al., 2021; Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021; Perl et al., 2015; Kamei et al., 2016; Hoang et al., 2019): *development-process* and *cross-project*. The details of data splitting for both settings are shown in Table 2.

In the *development- (dev.-) process* setting considering the impact of time on the JIT-VD approaches' performance, we follow the same time-aware evaluation procedure to construct the training data and testing data from the dataset as in the prior work (Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021; Perl et al., 2015; Hoang et al., 2019). Particularly, we divided the commits into those before and after time point t. The dangerous/safe commits before t were used for training, while the commits after t were used for evaluation. We selected a time point t to achieve a random training/test split ratio of 80/20 based on time. Specifically, for the *dev.-process* setting, the commits from Aug 1998 to Mar 2017 are used for training and the commits from Apr 2017 to Aug 2022 are for evaluation. In total, the training/test split in the number of commits for this setting is 16,219/4055.

Similar to existing work (Kamei et al., 2016; Zeng et al., 2021), in the *cross-project* setting, we evaluate how well the approaches can learn to recognize dangerous commits in a set of projects and detect suspicious commits in the other set. Specifically, the commits in a fixed set of projects are used to train the approaches, the remaining set of commits is used for testing. For this setting, the whole set of projects is randomly split into 80% (402 projects) for training and 20% (104 projects) for testing. The training/test split in the number of commits for this setting is 16,890/3,384.

Table 3
Dev.-process: JIT-VD performance comparison.

	Precision	*Recall*	*F1*	*Accuracy*
VCCFinder	0.50	**0.75**	0.60	0.70
CC2Vec+DeepJIT	0.33	0.66	0.44	0.49
LA_Predict	0.59	0.58	0.59	0.75
JITLine	0.62	0.66	0.64	0.77
JITFine	0.62	0.73	0.67	0.78
CodeJIT	**0.78**	0.70	**0.74**	**0.85**

RQ2. Intrinsic Analysis. We also investigated the impact of the GNN model and the number of GNN layers on CodeJIT's performance. We used different variants of relational graph neural networks and their properties to study the impact of those factors on CodeJIT's performance.

RQ3. Change Representation Analysis. We studied the impacts of the code relations (syntactic structure and dependency) and unchanged code in CTGs on CodeJIT's performance. We used different variants of CTGs and measured the performance of CodeJIT with each variant.

RQ4. Sensitivity Analysis. We studied the impacts of the following factors on the performance of CodeJIT: training size and change complexity (the rate of the changed nodes over the total of nodes in CTGs). To systematically vary these factors, we gradually added more training data and varied the range of the change rate.

5.2.2. Metrics

The task of JIT-VD can be considered as a binary classification problem that takes a commit as input and classifies it to be label 1 (*i.e.*, vulnerable) or label 0 (*i.e.*, non-vulnerable) (Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021; Zeng et al., 2021; Perl et al., 2015; Hoang et al., 2019). To evaluate the JIT-VD approaches (RQ1–RQ4), we measure the classification metrics such as *classification accuracy*, *precision*, and *recall*, as well as *F1* which is a harmonic mean of precision and recall. Particularly, the classification accuracy (*accuracy* for short) is the fraction of the (dangerous and safe) commits which are correctly classified among all the tested commits. For detecting dangerous commits, *precision* is the fraction of correctly detected dangerous commits among the detected dangerous commits, while *recall* is the fraction of correctly detected dangerous commits among the dangerous commits. Formally $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$, where TP is the number of true positives, FP and FN are the numbers of false positives and false negatives, respectively. *F1* is calculated as $F1 = \frac{2\times precision\times recall}{precision+recall}$. These metrics are used to answer all four research questions RQ1-RQ4. For all of these metrics, the higher the value, the better the approach to detecting vulnerable commits.

6. Experimental results

6.1. JIT-VD performance comparison (RQ1)

6.1.1. JIT-VD performance comparison

Tables 3 and 4 show the performance of CodeJIT and the other JIT-VD approaches in the *dev.-process* and *cross-project* settings. CodeJIT significantly outperforms the state-of-the-art JIT-VD approaches in JIT-VD *F1* and classification *accuracy* in both settings.

In the *dev.-process* setting, the JIT-VD *F1* and *accuracy* of CodeJIT are significantly better than those of the baseline approaches by **10–68%** and **8–73%**, respectively. Meanwhile, CodeJIT's *recall* in the *dev.-process* setting is just slightly lower than those of VCCFinder (Perl et al., 2015) and JITFine (Ni et al., 2022), by up to 7%. The predictions of CodeJIT are much more precise than those of all the studied methods. Specifically, CodeJIT improves the *precision* of the other approaches by **15–136%** in this setting.

In the *cross-project* setting, CodeJIT consistently improves the state-of-the-art JIT-VD approaches by **23–65**% in JIT-VD *F1* and **17–77**%

Table 4
Cross-project: JIT-VD performance comparison.

	Precision	*Recall*	*F1*	*Accuracy*
VCCFinder	0.58	0.49	0.53	0.74
CC2Vec+DeepJIT	0.40	0.69	0.51	0.50
LA_Predict	0.66	0.48	0.56	0.72
JITLine	0.57	0.72	0.64	0.69
JITFine	0.68	0.69	0.68	0.76
CodeJIT	**0.90**	**0.80**	**0.84**	**0.89**

Table 5
False negative and *false positive* rates of JIT-VD approaches.

	Dev.-process		*Cross-project*	
	FN	*FP*	*FN*	*FP*
VCCFinder	**0.25**	0.50	0.51	0.42
CC2Vec+DeepJIT	0.34	0.67	0.31	0.60
LA_Predict	0.42	0.40	0.51	0.34
JITLine	0.34	0.38	0.28	0.43
JITFine	0.27	0.38	0.31	0.32
CodeJIT	0.30	**0.22**	**0.20**	**0.10**

FN: False Negative Rate; *FP*: False Positive Rate.

in classification *accuracy*. CodeJIT achieves the *recall* of **0.8**, which is significantly better than the corresponding rates achieved by all the other approaches, with improvements ranging from **11–66%**. This means that CodeJIT can effectively detect **8/10** dangerous commits. Especially, the *precision* of CodeJIT in this setting is **90%**. In other words, **9/10** suspicious commits detected by CodeJIT actually are dangerous ones. Meanwhile, the corresponding figures of the existing methods are 4.0–6.8.

Moreover, the results in Tables 3 and 4 show that CodeJIT performs stably in both settings. Particularly, CodeJIT consistently achieves the highest overall performance (*F1* and *accuracy*) among the JIT-VD approaches. CodeJIT also maintains the same pattern of improvement for both *precision* and *recall* when switching between the *dev.-process* setting and *cross-project* setting. For most of the other approaches, an increase in *precision* is typically accompanied by a decrease in *recall* when switching from one setting to the other. For example, in the *dev.-process* setting, VCCFinder can find much more dangerous commits (higher *recall*), yet raise more false alarms (lower *precision*) compared to this approach in the *cross-project* setting. In other words, these approaches cannot detect dangerous commit more effectively and more precisely at the same time.

Table 5 shows the *false negative* rates (*FN*) and *false positive* rates (*FP*) of CodeJIT and the baselines. Particularly, the *FN* refers to the fraction of dangerous commits that cannot be detected by the approaches, while the *FP* refers to the ratio of safe commits incorrectly predicted as dangerous. The lower the *FN* and *FP*, the better the JIT-VD approach. As seen, VCCFinder obtains the lowest *FN*, in *dev.-process* setting. However, its *FP* is relatively high. This could lead to an unnecessary investigation of safe commits. Meanwhile, CodeJIT obtains the lowest *FP* among the JIT-VD approaches in both *dev.-process* and *cross-project*. Especially, CodeJIT obtains the best performance in both *FN* and *FP*. *These illustrate that* CodeJIT *not only misses the fewest dangerous commits (low FN) but also wastes the least unnecessary investigation in safe commits (low FP).*

We also evaluated the JIT-VD approaches on imbalanced datasets with various vulnerability ratios (vulnerable commits:non-vulnerable commits) by gradually adding more non-vulnerable commits to the dataset. Indeed, all JIT-VD approaches suffer from decreased performance in imbalanced dataset settings. However, in these experimental scenarios, CodeJIT consistently obtained better results than the other approaches. On the dataset where the vulnerability ratio is 1:30, the F1 scores of JITFine and JITLine are only about 0.13, while CodeJIT's F1 is 0.37, which is 183% better than those of JITFine and JITLine. In our experiments, we found that the relative improvements of CodeJIT over the others constantly increase when the imbalance problem is more severe. Indeed, the improvement rates in F1 of CodeJIT over JITFine and JITLine in the balanced case is about only 10%–15%, while CodeJIT's improvement rates in F1 are about 100% and 183% with the vulnerability ratios of 1:10 and 1:30, respectively.

> **Answer for RQ1**: For both settings, CodeJIT can precisely detect a larger number of dangerous commits compared to the state-of-the-art approaches. This confirms our code-centric strategy in detecting vulnerabilities at the commit-level.

6.1.2. Result analysis

In our experiments, we found that 69/188 fixing commits, which cause other issues, are correctly detected by CodeJIT. Meanwhile, the corresponding figures of JITLine and LA_Predict are 58 and 41, respectively. Fig. 8(a) shows a commit fixing a bug, yet causing another problem in *FFmpeg*. Specifically, to prevent the out-of-bound problem, `ch_data->bs_num_env` is set to a valid value if it is larger than the allowance (line 643 in the new version, Fig. 8(a)). However, changing the value of this variable without appropriately checking causes another serious problem, which is later fixed by the commit `87b08ee`. The dangerous commit in Fig. 8(a) is correctly detected by CodeJIT, while the other methods misclassify this commit as a *safe* one.

Specifically, the approaches leveraging commit message such as JITLine and JITFine could be misled by special keywords such as "*Fixes*" in the commit message: "*avcodec/aacsbr_template: Do not leave bs_num_env invalid* ***Fixes*** *out of array read* ***Fixes****: 1349/clusterfuzz...*". Zeng et al. (2021) show that the message of a commit could provide the intention of the commit. However, developers often unintentionally bundle unrelated changes with different purposes (*e.g.*, bug fix and refactoring) in a single commit (tangled commit) (Dias et al., 2015). Multiple studies have shown that fixing a bug could cause another bug (Guo et al., 2010; Purushothaman and Perry, 2005). Moreover, in practice, commit messages are often poor-quality, even empty (Tian et al., 2022). This makes the application of commit messages for JIT-VD much less reliable and causes incorrect predictions. Meanwhile, LA_Predict considers only the line-added-number to evaluate the suspiciousness based on the hypothesis that the larger the line-added-number, the higher the probability of being defective (Zeng et al., 2021). In fact, the commit in the example above contains only two added lines of code, which leads to the incorrect prediction of LA_Predict.

By capturing the changes in the code syntactic structure expressed in the commit's CTG, CodeJIT is able to recognize that the value of `ch_data->bs_num_env` is dangerously changed (line 643 of the new version) after checking the allowed bound (line 639). In addition, by analyzing the dependencies, CodeJIT can identify that the newly changed value is a magic number (2) and has no relation with the checked bound (4), which means that the corresponding commit still does not guarantee the safety of code after being executed. Then, this issue is fixed by commit `87b08ee` (Fig. 8(b)) by using a temporary variable (`bs_num_env`) instead of directly using and defining `ch_data->bs_num_env`. As expected, CodeJIT correctly detects both these two commits.

In our experiments, the observed better performance of CodeJIT in the *cross-project* setting compared to the *dev.-process* setting can be attributed to our dataset division strategy. In the *cross-project* setting, the model benefits from a broader scope of knowledge during training and testing. When dividing the dataset by project, commits from a project p in the training set encompass all its historical commits, allowing CodeJIT to learn a comprehensive understanding of vulnerabilities during the whole development process of p. In the testing set, a commit c from a different project p' can be classified based on the knowledge acquired from the commits in p, even those occurring after c. This enables CodeJIT to leverage knowledge from commits beyond the current time point, leading to improved performance in the *cross-project* setting.

Overall, *properly utilizing the code-related aspects of commits could help* CodeJIT *precisely understand the semantics of the changes and effectively learn to differentiate safe/dangerous commits, thus significantly improving the JIT-VD performance*.

```
633  633    case FIXFIX:
...            //...
639  639        if (ch_data->bs_num_env > 4) {
640  640            av_log(ac->avctx, AV_LOG_ERROR,
641  641                   "Invalid bitstream,...\n",
642  642                   ch_data->bs_num_env);
     643  +         ch_data->bs_num_env = 2;
643  644            return -1;
644  645        }
```

(a) Commit a8ad83b fixing a vulnerability in *FFmpeg* but causing another

```
633  634    case FIXFIX:
634       -    ch_data->bs_num_env  = 1 << get_bits(gb, 2);
...            //...
639       -    if (ch_data->bs_num_env > 4) {
     635  +    bs_num_env = 1 << get_bits(gb, 2);
     636  +    if (bs_num_env > 4) {
640  637            av_log(ac->avctx, AV_LOG_ERROR,
641  638                   "Invalid bitstream, too many SBR...\n",
642       -                 ch_data->bs_num_env);
643       -            ch_data->bs_num_env = 2;
     639  +                 bs_num_env);
644  640            return -1;
645  641        }
```

(b) Commit 87b08ee fixing the vulnerability in Figure 8a

Fig. 8. Example of a dangerous commit correctly detected by CodeJIT.

Table 6
Impact of GNN models on JIT-VD Performance.

	Precision	*Recall*	*F1*	*Accuracy*
RGCN	0.78	0.70	0.74	0.85
FastRGCN	0.77	0.71	0.74	0.85
RGAT	0.82	0.66	0.73	0.85

6.2. Intrinsic analysis (RQ2)

6.2.1. Impact of GNN model

To investigate the impact of different GNN models on the JIT-VD performance, we compared three variants of relational graph networks: CodeJIT with RGCN (Schlichtkrull et al., 2018), RGAT (Busbridge et al., 2019), and FastRGCN (Chen et al., 2018). In this experiment, we used the *sum* readout function and two GNN layers, and applied for the *dev.-process* setting. The results of those three variants is shown in Table 6. As expected, CodeJIT obtains quite stable performance with about 0.74% in *F1* and 0.85 in *accuracy*. Moreover, while CodeJIT obtains quite similar *precision* and *recall* with both RCGN and FastRGCN, it achieves the highest *precision*, yet lowest *recall* with RGAT. Indeed, the attention mechanism enables RGAT to focus on the important features of the neighboring nodes in each relation for computing the features of the graph nodes. This helps RGAT precisely detect dangerous commits and obtains the highest *precision*. However, the attention mechanism in the RGAT model may be causing it to prioritize certain features, leading to higher *precision* but lower *recall*. This can happen if the attention mechanism is too focused on specific features and fails to capture the broader context of the data. Thus, *light-weight relational graph neural networks such as RCGN (Schlichtkrull et al., 2018) and FastRGCN (Chen et al., 2018) should be applied to achieve cost-effective JIT-VD performance*.

6.2.2. Impact of number of GNN layers

To study the impact of the number of GNN layers on the JIT-VD performance of CodeJIT, we varied the number of GNN layers in JIT-VD model from 1 to 5. In this experiment, we used RGAT for the JIT-VD model and the *dev.-process* setting. As seen in Fig. 9, the JIT-VD *F1* and classification *accuracy* of CodeJIT are significantly better when increasing the number of RGCN layers in the JIT-VD model from one to two. With two RGCN layers, the model represents each node in CTGs from the 2-hop neighborhood. Meanwhile, each node is represented by considering only its neighbors by the model with one RGCN layer. Thus, by using more information to represent each node in CTGs, the model detected dangerous commits more precisely. Indeed, the precision of CodeJIT is remarkably improved by 67% with two RGCN layers. As expected, the more complex node representation mechanism slightly lowers the recall by 7%. However, considering longer 2-hop neighborhood to represent nodes in CTGs causes downgrades in the detection performance. Particularly, the performance CodeJIT gracefully decreases when the number of GNN layers increases from 2 to 5. This could be because for a node, the further information collected from the other nodes could bring noises in representing the node. Additionally, with a larger number of layers and a longer neighborhood relation, the set of the shared neighbors which are used to represent nodes could be larger. Thus, the different nodes could be represented similarly. Consequently, the discriminating performance of GNN model is reduced. Moreover, the model with more layers is much more complicated. Thus, *we use two GNN layers to ensure the best performance and simplicity*.

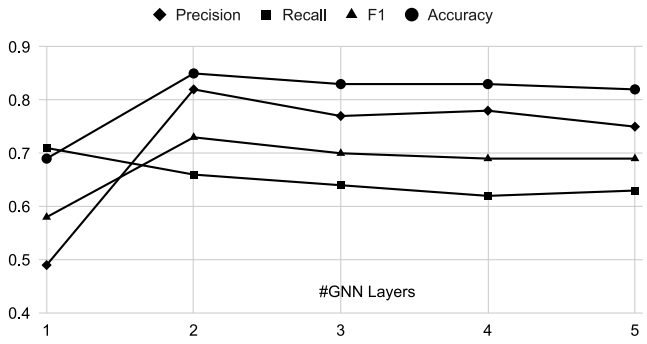

Fig. 9. Impact of the number of GNN layers.

Table 7
Impact of graph readout function.

	Precision	*Recall*	*F1*	*Accuracy*
Max	0.93	0.73	0.81	0.88
Average	0.92	0.73	0.81	0.88
Sum	0.95	0.70	0.80	0.87

6.2.3. Impact of graph readout function

To evaluate the impact of the different readout functions in forming the whole feature vector of a graph, we created different variants of CodeJIT with *Max*, *Average*, and *Sum* function to aggregate node features. In this experiment, we used the RGAT model, with two layers for the *cross-project* setting. As shown in Table 7, CodeJIT's performance is not significantly affected when the aggregation function is changed. Specifically, the average *F1* and *accuracy* are 0.81 and 0.88, respectively. *As the performance of* CodeJIT *is stable with different readout functions, either Max, Average, or Sum could be used in our JIT-VD model.*

> **Answer for RQ2**: The JIT-VD model's properties/components impact CodeJIT's effectiveness differently. The GNN model and graph readout function slightly impact the performance, while the number of GNN layers could significantly impact CodeJIT's performance.

6.3. Change representation analysis (RQ3)

6.3.1. Unchanged code's role analysis

To investigate the contribution of the related unchanged code in CTGs, we used two variants of CTG: one considering both changed code and unchanged code (CTG), the other considering only changed

Table 8
Impact of the unchanged code in CTGs on CodeJIT's performance.

	Precision	*Recall*	*F1*	*Accuracy*
CodeJIT$_{CTG}$	0.95	0.70	0.80	0.87
CodeJIT$_{\hat{CTG}}$	0.79	0.61	0.69	0.79

Table 9
Impact of the code relations on CodeJIT's performance.

	Precision	*Recall*	*F1*	*Accuracy*
Syn. Struct.	0.72	0.66	0.69	0.82
Prog. Depend.	0.70	0.69	0.69	0.81
Syn. Struct. +Prog. Depend.	0.82	0.66	0.73	0.85

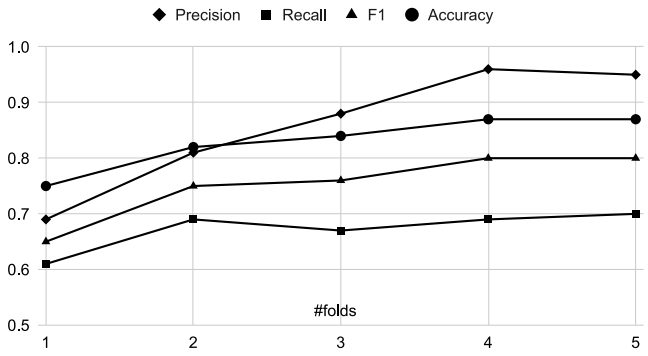

Fig. 10. Impact of training data size.

code ($\hat{CTG}$). Table 8 shows the JIT-VD performance of CodeJIT using the two CTG variants: CodeJIT$_{CTG}$ and CodeJIT$_{\hat{CTG}}$. Note that, in this experiment, we applied for the *cross-project* setting and used the same RGAT model, with two layers, and sum readout function for both CTG variants.

As seen, additionally considering the related unchanged code along with changed code in CTGs significantly improves the JIT-VD performance of CodeJIT using CTGs with only changed code. Particularly, CodeJIT achieves better *precision* and *recall* by 20% and 15%, respectively. The related unchanged code provides valuable information and helps the model not only understand code changes more precisely but also discover more vulnerability patterns. The number of dangerous commits detected by CodeJIT$_{CTG}$ but not detected by CodeJIT$_{\hat{CTG}}$ (149 commits) nearly triples the corresponding figure by CodeJIT$_{\hat{CTG}}$ but not detected by CodeJIT$_{CTG}$ (only 52 commits). This experimentally confirms our observation 2 on the important role of related unchanged code for understanding code changes in JIT-VD.

6.3.2. Code relation analysis

To analyze the impact of the relations in CTGs on the JIT-VD performance, we used different variants of CTGs: $\mathcal{R} = \{syntactic\ structure\}$, $\mathcal{R} = \{program\ dependency\}$, and $\mathcal{R} = \{syntactic\ structure, program\ dependency\}$. This experiment uses the same setting for the JIT-VD model (RGAT model having two layers, and *sum* readout function) for all the variants, and applied in the *dev.-process* setting.

In Table 9, CodeJIT performs quite stably when $\mathcal{R} = \{syntactic\ structure\}$ and $\mathcal{R} = \{dependency\}$, *i.e.*, *F1 = 0.69* in both cases. When combining both relations in CTGs, CodeJIT detects the dangerous commits significantly more precisely while its *recall* remains stable. It is reasonable because considering both *syntactic structure* and *program dependency* in CTGs gains the information that is necessary for the model to determine if a commit is suspiciously dangerous. As a result, both the JIT-VD *F1* and *accuracy* increase when $\mathcal{R} = \{syntactic\ structure, programdependency\}$.

Since several existing studies consider the execution flow (*i.e.*, control flow) relation between program elements in detecting vulnerabilities (Gascon et al., 2013; Sparks et al., 2007), we additionally consider $\mathcal{R} = \{syntactic\ structure, programdependency, exec.flow\}$ in CTGs to evaluate the expandability of our proposed code change representation. We found that additionally considering the execution relation in CTGs, $\mathcal{R} = \{syntactic\ structure, program\ dependency, exec.flow\}$, decreases the performance of CodeJIT by 1.4% in *F1*. As expected, in this case, the *precision* is 4.8% better than that of CodeJIT with $\mathcal{R} = \{syntactic\ structure, programdependency\}$. Nevertheless, the changes in the execution flow (*i.e.*, control flow), that are required for determining dangerous commits, could be covered by the changes in the dependency relation via control dependencies. This redundancy hurts CodeJIT's recall 6.1%. Moreover, this conditional consideration increases the complexity of CTGs and the cost for training and testing the JIT-VD model. Thus, in this work, we use $\mathcal{R} = \{syntactic\ structure, programdependency\}$ in CTGs for the best performance and efficiency of CodeJIT.

Answer for RQ3: Besides changed code in CTGs, the related unchanged code significantly contributes to the effectiveness of CodeJIT in detecting dangerous commits. Additionally, incorporating *syntactic structure* and *program dependency* relations in CTGs enhances the JIT-VD performance of CodeJIT.

6.4. Sensitivity analysis (RQ4)

6.4.1. Impact of training size

To measure the impact of training data size on CodeJIT's performance, we use the *cross-project* setting, in which the training set is randomly separated into 5 folds. We gradually increased the sizes of the training dataset by adding one fold at a time until all 5 folds were added for training. The results of this experiment are shown in Fig. 10. As expected, CodeJIT's performance is improved when expanding the training dataset. Especially, the precision significantly grows by 38% when increasing the training size from 1 fold to 5 folds. The reason is that with larger training datasets, CodeJIT has observed more and then performs better. At the same time, training with a larger dataset costs more time. Particularly, the training time of CodeJIT with 5 folds is six times more than the training time with only 1 fold.

6.4.2. Impact of change complexity

In this experiment, we investigated the sensitivity of CodeJIT's performance on the input complexity in change rates which are measured by the ratio of the number of changed nodes over the total number of nodes of each CTG (Fig. 11). As seen, there are much fewer commits (lower distribution - Distr.) with higher complexity levels. Meanwhile, the performance of CodeJIT in both *F1* and *accuracy* is quite stable when handling commits in different change rates. Specifically, the *F1* of CodeJIT gracefully grows from 75% to 89%, and the *accuracy* is in the range of 86% to 89% when increasing the change rate.

We also studied the sensitivity of CodeJIT's inference on the input complexity in the number of changed lines. As seen in Fig. 12, the *recall* of CodeJIT increases and its *precision* is relatively stable with the commits of different complexity levels. The reason could be that using CTGs and the graph-based JIT-VD model makes CodeJIT capture precisely the patterns of vulnerabilities at the commit-level. Although the number of commits significantly decreases, and there are fewer vulnerability patterns when the complexity is higher, with more significant amounts of changed code, CodeJIT has more information to determine if commits are dangerous. Thus, CodeJIT still maintains its performance with large commits.

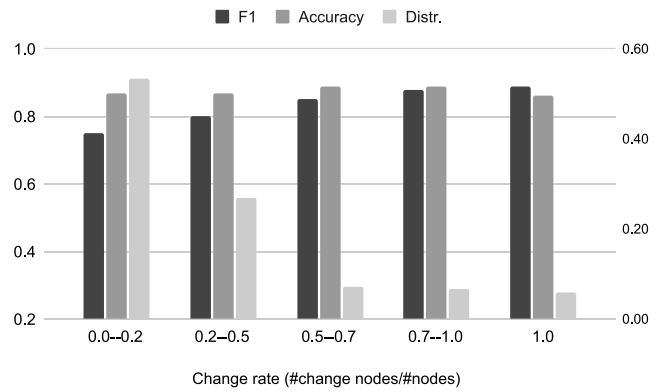

Fig. 11. Impact of change complexity (left axis: *F1* & *accuracy*; right axis: Distr.).

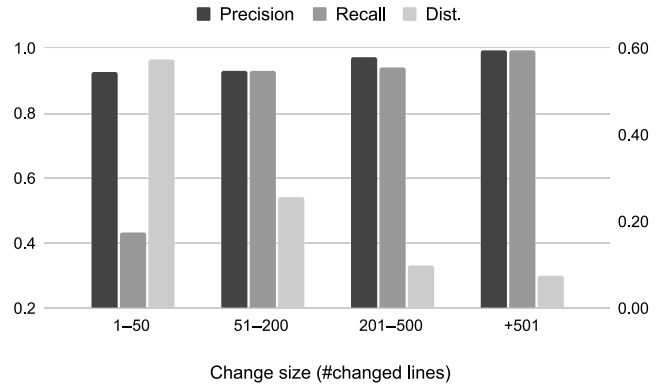

Fig. 12. Impact of change size (left axis: precision & recall; right axis: distribution).

Answer for RQ4: The JIT-VD performance of CodeJIT improves when CodeJIT is trained on a larger dataset. The performance of CodeJIT in both *F1* and *accuracy* is stable when classifying commits with different change complexity levels in the rate of changed nodes.

6.5. Time complexity (RQ5)

All our experiments were run on a server running Ubuntu 18.04 with an NVIDIA Tesla P100 GPU. On average, CodeJIT took less than a second to construct a CTG. To train the JIT-VD model, CodeJIT took about 5 h for 50 epochs. Additionally, CodeJIT with FastRGCN spent 0.75 s to classify whether a commit is suspicious or not.

As expected, the more complicated the JIT-VD model, the longer the training time. For a fold of data, the 1-layer RGAT model took about 40 min, while these figures for the 3-layer RGAT model and 5-layer RGAT models are 1.5 times and 1.8 times larger, respectively. In our experiments, we found that the larger the training dataset, the longer the training time. Specifically, the training time of five folds of data is six times more than that of only one fold. Meanwhile, the classification time is not quite different among the variants of CodeJIT. For example, with FastRGCN, the average classification time is 0.75 s, while CodeJIT with RGCN took 1.42 s to classify a commit.

In practice, we need to trade-off between performance and the scalability of the approach. The more complicated the architecture and the larger the training data size, the better the performance the models could obtain. However, such a model also needs longer training and inferring time. Moreover, similar to the other GNN-based approaches, CodeJIT also faces the challenges of dealing with large graphs. The reason is that processing large graphs can be computationally expensive and require substantial memory resources.

6.6. Threats to validity

The main threats to the validity of our work consist of internal, construct, and external threats.

Threats to internal validity include the hyperparameter settings of all of the baseline models as well as the ones chosen for CodeJIT. To reduce this threat, we systematically varied the setting of CodeJIT to study its performance (Section 6.2) and reused the implementation with the same setting in the papers of the baseline approaches (Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021; Zeng et al., 2021; Perl et al., 2015; Hoang et al., 2019). A threat may come from the method used to construct relational code graph and identify the relation between program entities. To reduce this threat, we use Joern (Yamaguchi et al., 2014) code analyzer, which is widely used in existing studies (Ding et al., 2022b; Li et al., 2018, 2021a; Hin et al., 2022). Another threat mainly lies in the correctness of the implementation of our approach. To reduce such threat, we carefully reviewed our code and made it public (Nguyen et al., 2024) so that other researchers can double-check and reproduce our experiments.

Threats to construct validity relate to the suitability of our evaluation procedure. We used *precision*, *recall*, *F1*, and classification *accuracy*. They are the widely-used evaluation measures for just-in-time defect detection (Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021; Zeng et al., 2021) and just-in-time vulnerability detection (Perl et al., 2015; Yang et al., 2017). In addition, the construction of our dataset is another threat which is also suffered by related studies (Zeng et al., 2021; Perl et al., 2015; Chakraborty et al., 2021). Specifically, the commits triggering/contributing to vulnerabilities are considered as dangerous ones. To reduce the impact of this threat, we focus on the vulnerability triggering commits as the dangerous commits instead of vulnerability contributing commits. Also, to precisely blame vulnerable statements while collecting dangerous commits, we consider program dependencies instead of surrounding statements, as mentioned in Section 5.1. Another threat is considering the vulnerability-fixing commits as safe ones because these commits could contribute to vulnerabilities. To reduce this threat, only the fixing commits which are not VCCs are considered as safe. A threat may come from the vulnerability ratio in our dataset, which might not reflect well the vulnerability ratio in practice. However, we applied the same data for all the JIT-VD approaches, and this ratio has been applied in several existing studies (Zhou et al., 2019; Wu et al., 2022). To mitigate this threat, we also evaluated the performance of the approaches and the improvement trend of CodeJIT compared to the other approaches on the datasets with various levels of imbalance. Another threat may come from the evaluation in controlled environments. To reduce this threat, we evaluated the JIT-VD approaches in two settings (*dev. process* and *cross-project*) and investigated CodeJIT's performance in various scenarios and settings.

Threats to external validity mainly lie in the selection of graph neural network models used in our experiments. To mitigate this threat, we select the representative models which are well-known for NLP and SE tasks (Schlichtkrull et al., 2018; Busbridge et al., 2019; Chen et al., 2018). Moreover, our experiments are conducted on only the code changes of C/C++ projects. Thus, the results could not be generalized for other programming languages. In our future work, we plan to conduct more experiments to validate our results in other languages.

7. Related work

CodeJIT relates to the work on **just-in-time vulnerability detection** (Perl et al., 2015; Yang et al., 2017). VCCFinder (Perl et al., 2015) and VulDigger (Yang et al., 2017) are the classification models combining code-metrics, commit messages, and expert features for JIT-VD. Meanwhile, the work on **just-in-time bug detection** (Ni et al., 2022; Pornprasit and Tantithamthavorn, 2021; Zeng et al., 2021; Hoang et al., 2019) could also be applied to detect vulnerabilities at the commit-level. DeepJIT (Hoang et al., 2019) automatically extracts features

from commit messages and changed code and uses them to identify defects. Pornprasit et al. propose JITLine, a simple but effective just-in-time defect prediction approach. JITLine utilizes the expert features and token features using bag-of-words from commit messages and changed code to build a defect prediction model with random forest classifier. LAPredict (Zeng et al., 2021) is a defect prediction model by leveraging the information of "lines of code added" expert feature with the traditional logistic regression classifier. Recently, Ni et al. introduced JITFine (Ni et al., 2022) combining the expert features and the semantic features which are extracted by CodeBERT (Feng et al., 2020) from changed code and commit messages.

Different from all prior studies in just-in-time bug/vulnerability detection, our work presents the first study centralizing the role of code change semantics in detecting vulnerabilities at the commit level. In CodeJIT, the code change meaning is the only deciding factor in examining the suspiciousness of commits. Additionally, CodeJIT is the first work representing code changes as graphs and applying GNN for JIT-VD. Our results experimentally show that our strategy is more appropriate for JIT-VD.

Vulnerability detection is also a critical part to be discussed. Various methods have been proposed to determine if a code component (component, file, function/method, or statement/line) is vulnerable. Recently, several deep-learning based approaches have been introduced (Lin et al., 2020, 2017; Li et al., 2018; Duan et al., 2019; Zhou et al., 2019; Chakraborty et al., 2021; Cao et al., 2022; Cheng et al., 2022; Vo and Nguyen, 2023). Devign (Zhou et al., 2019) leverages code property graph (CPG) to build their graph-based vulnerability prediction model. VulDeePecker (Li et al., 2018) and SySeVR (Li et al., 2021b) introduce tools to detect *slice-level* vulnerabilities, which are more fine-grained. IVDetect (Li et al., 2021a), which is a graph-based neural network model, is proposed to detect vulnerabilities at the function level and use a model interpreter to identify vulnerable statements in the detected suspicious functions. LineVul (Fu and Tantithamthavorn, 2022) and LineVD (Hin et al., 2022) apply CodeBERT in their own way and have been shown that they are more effective than IVDetect in detecting vulnerable functions and lines/statements. VelVet (Ding et al., 2022b) builds graph-based models to detect vulnerable statements.

Although using syntactic structure and program dependencies to *represent code* has been employed in prior vulnerability detection studies, such as Devign (Zhou et al., 2019) and IVDetect (Li et al., 2021a), our work introduces a novel method to *represent code changes* in the context of JIT-VD. Additionally, while applying GNN for vulnerability detection at the file level or method level is not novel, to the best of our knowledge, the application of GNN specifically for vulnerability detection at the commit level has remained largely unexplored. In our work, we have introduced modifications to embed nodes and edges, adapting GNN to suit our JIT-VD task effectively. Moreover, those existing approaches focus on detecting vulnerabilities at the release time, while CodeJIT is a just-in-time vulnerability detection. CodeJIT and the release-time approaches could complement to support developers in ensuring software quality in the development process.

Several **learning-based approaches** have been proposed for specific SE tasks including code suggestion (Nguyen et al., 2020; Hindle et al., 2016; Allamanis et al., 2016; Nguyen et al., 2023a), program synthesis (Amodio et al., 2017; Gvero and Kuncak, 2015), pull request description generation (Hu et al., 2018; Liu et al., 2019), code summarization (Iyer et al., 2016; Mastropaolo et al., 2021; Wan et al., 2018), code clones (Li et al., 2017), fuzz testing (Godefroid et al., 2017), bug detection (Li et al., 2019), bug fix identification (Nguyen et al., 2023b), and program repair (Jiang et al., 2021; Ding et al., 2020).

8. Conclusion

In this paper, we propose CodeJIT, a novel code-centric approach for just-in-time vulnerability detection (JIT-VD). Our key idea is that *the meaning of the changes in source code caused by a commit is the direct and deciding factor for assessing the commit's riskiness.* We design a novel graph-based multi-view code change representation of the changed code of commits in relation to the unchanged code. Particularly, a graph-based JIT-VD model is developed to capture the patterns of dangerous/safe (benign) commits and detect vulnerabilities at the commit level. On a large public dataset of dangerous and safe commits, our results show that CodeJIT significantly improves the state-of-the-art JIT-VD methods by up to 66% in *Recall*, 136% in *Precision*, and 68% in *F1*. Moreover, CodeJIT correctly classifies nearly 9/10 of dangerous/safe commits.

Ethical considerations are paramount in our JIT-VD research, emphasizing responsible application for enhanced cybersecurity. Our core principle is empowering developers with proactive tools to identify and mitigate vulnerabilities, promoting overall system resilience. We acknowledge ethical aspects extend beyond technology to include concerns about potential misuse. We actively discourage unauthorized or malicious use, aligning with our ethical framework. We commit to ongoing collaboration with the security community, responsibly sharing insights to collectively safeguard digital systems. To further minimize the ethical concerns, we constructed the dataset and conducted experiments on the vulnerabilities that have been publicly disclosed. We prioritize transparency and accountability, mitigating the risk of unintentional exposure of sensitive data. This also ensures the utmost respect for privacy and safeguards against reputational harm to any individuals or organizations involved. Our research seeks to contribute to cybersecurity advancements while maintaining the highest ethical standards in vulnerability detection research. Addressing ethical aspects ensures our research contributes positively to cybersecurity, prioritizing digital ecosystem well-being and security principles.

Our future work will focus on advancing just-in-time vulnerability detection at the node level. This refined level of granularity will significantly enhance our capacity to pinpoint vulnerabilities within the code. This fine-grained approach will not only provide more precise vulnerability identification but also offer developers the means to expedite vulnerability localization and resolution, ultimately strengthening software security.

CRediT authorship contribution statement

Son Nguyen: Conceptualization, Formal analysis, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Thu-Trang Nguyen:** Conceptualization, Investigation, Methodology, Writing – original draft, Writing – review & editing. **Thanh Trong Vu:** Investigation, Software, Validation, Data curation. **Thanh-Dat Do:** Data curation, Investigation, Software, Validation. **Kien-Tuan Ngo:** Data curation, Investigation, Methodology, Software, Validation. **Hieu Dinh Vo:** Conceptualization, Formal analysis, Investigation, Methodology, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

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

Data availability

Data will be made available on request.

Acknowledgments

This research is supported by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2023.14.

References

Allamanis, M., Peng, H., Sutton, C., 2016. A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning. PMLR, pp. 2091–2100.

Amodio, M., Chaudhuri, S., Reps, T.W., 2017. Neural attribute machines for program generation. CoRR.

Bhandari, G., Naseer, A., Moonen, L., 2021. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 30–39.

Braz, L., Aeberhard, C., Çalikli, G., Bacchelli, A., 2022. Less is more: Supporting developers in vulnerability detection during code review. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1317–1329.

Busbridge, D., Sherburn, D., Cavallo, P., Hammerla, N.Y., 2019. Relational graph attention networks. arXiv preprint arXiv:1904.05811.

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C., 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, pp. 1456–1468.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2021. Deep learning based vulnerability detection: Are we there yet. IEEE Trans. Softw. Eng..

Checkmarx, 2022. Checkmarx. https://checkmarx.com/. (Online; accessed 19 October 2022).

Chen, J., Ma, T., Xiao, C., 2018. Fastgcn: Fast learning with graph convolutional networks via importance sampling. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.

Cheng, X., Zhang, G., Wang, H., Sui, Y., 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2022, Association for Computing Machinery, New York, NY, USA, pp. 519–531.

Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S., 2015. Untangling fine-grained code changes. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 341–350.

Ding, Z., Li, H., Shang, W., Chen, T.-H.P., 2022a. Can pre-trained code embeddings improve model performance? Revisiting the use of code embeddings in software engineering tasks. Empir. Softw. Eng. 27 (3), 1–38.

Ding, Y., Ray, B., Devanbu, P., Hellendoorn, V.J., 2020. Patching as translation: The data and the metaphor. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 275–286.

Ding, Y., Suneja, S., Zheng, Y., Laredo, J., Morari, A., Kaiser, G., Ray, B., 2022b. VELVET: A novel ensemble learning approach to automatically locate VulnErable statements. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering. IEEE, pp. 959–970.

Dong, J., Lou, Y., Zhu, Q., Sun, Z., Li, Z., Zhang, W., Hao, D., 2022. FIRA: Fine-grained graph-based code change representation for automated commit message generation. In: Proceedings of the 44th International Conference on Software Engineering. pp. 970–981.

Duan, X., Wu, J., Ji, S., Rui, Z., Luo, T., Yang, M., Wu, Y., 2019. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence. pp. 4665–4671.

Elder, S., Zahan, N., Shu, R., Metro, M., Kozarev, V., Menzies, T., Williams, L., 2022. Do I really need all this work to find vulnerabilities? Empir. Softw. Eng. 27 (6), 1–78.

Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 508–512.

Fan, Y., Xia, X., Da Costa, D.A., Lo, D., Hassan, A.E., Li, S., 2019. The impact of mislabeled changes by szz on just-in-time defect prediction. IEEE Trans. Softw. Eng. 47 (8), 1559–1586.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics, Online, pp. 1536–1547.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Programm. Lang. Syst. (TOPLAS) 9 (3), 319–349.

Flawfinder, 2022. Flawfinder. https://dwheeler.com/flawfinder/. (Online; accessed 19 October 2022).

Fu, M., Tantithamthavorn, C., 2022. LineVul: A transformer-based line-level vulnerability prediction. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories. MSR, IEEE Computer Society, Los Alamitos, CA, USA, pp. 608–620.

Gascon, H., Yamaguchi, F., Arp, D., Rieck, K., 2013. Structural detection of Android malware using embedded call graphs. In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security. pp. 45–54.

Godefroid, P., Peleg, H., Singh, R., 2017. Learn&fuzz: Machine learning for input fuzzing. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 50–59.

Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B., 2010. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1. pp. 495–504.

Gvero, T., Kuncak, V., 2015. Synthesizing Java expressions from free-form queries. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 416–432.

Hin, D., Kan, A., Chen, H., Babar, M.A., 2022. LineVD: Statement-level vulnerability detection using graph neural networks. In: IEEE/ACM 19th International Conference on Mining Software Repositories. MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, IEEE, pp. 596–607.

Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131.

Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N., 2019. DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 34–45.

Hoang, T., Kang, H.J., Lo, D., Lawall, J., 2020. Cc2vec: Distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 518–529.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension. ICPC, IEEE, pp. 200–20010.

Iannone, E., Guadagni, R., Ferrucci, F., De Lucia, A., Palomba, F., 2022. The secret life of software vulnerabilities: A large-scale empirical study. IEEE Trans. Softw. Eng..

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 2073–2083.

Jiang, N., Lutellier, T., Tan, L., 2021. CURE: Code-aware neural machine translation for automatic program repair. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 1161–1173.

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E., 2016. Studying just-in-time defect prediction using cross-project models. Empir. Softw. Eng. 21, 2072–2106.

Kang, J., Ryu, D., Baik, J., 2021. Predicting just-in-time software defects to reduce post-release quality costs in the maritime industry. Softw. - Pract. Exp. 51 (4), 748–771.

Kim, S., Woo, S., Lee, H., Oh, H., 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 595–614.

Krasner, H., 2021. The cost of poor software quality in the US: A 2020 report. Proc. Consortium Inf. Softw. QualityTM (CISQTM).

Li, L., Feng, H., Zhuang, W., Meng, N., Ryder, B., 2017. Cclearner: A deep learning-based clone detection approach. In: International Conference on Software Maintenance and Evolution. IEEE, pp. 249–260.

Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., et al., 2022. Automating code review activities by large-scale pre-training. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1035–1047.

Li, Y., Wang, S., Nguyen, T.N., 2021a. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303.

Li, Y., Wang, S., Nguyen, T.N., Van Nguyen, S., 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. ACM Programm. Lang. 3 (OOPSLA), 1–30.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: An automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. pp. 201–213.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021b. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput..

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In: 25th Annual Network and Distributed System Security Symposium. The Internet Society.

Lin, G., Wen, S., Han, Q.-L., Zhang, J., Xiang, Y., 2020. Software vulnerability detection using deep neural networks: A survey. Proc. IEEE 108 (10), 1825–1848.

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

Liu, S., Gao, C., Chen, S., Nie, L.Y., Liu, Y., 2020. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. IEEE Trans. Softw. Eng. 48 (5), 1800–1817.

Liu, Z., Xia, X., Treude, C., Lo, D., Li, S., 2019. Automatic generation of pull request descriptions. In: 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 176–188.

Lomio, F., Iannone, E., De Lucia, A., Palomba, F., Lenarduzzi, V., 2022. Just-in-time software vulnerability detection: Are we there yet? J. Syst. Softw. 111283.

Mastropaolo, A., Scalabrino, S., Cooper, N., Palacio, D.N., Poshyvanyk, D., Oliveto, R., Bavota, G., 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 336–347.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. In: Bengio, Y., LeCun, Y. (Eds.), 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings.

Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A., 2007. Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 529–540.

Ngo, K.-T., Do, D.-T., Nguyen, T.-T., Vo, H.D., 2021. Ranking warnings of static analysis tools using representation learning. In: 2021 28th Asia-Pacific Software Engineering Conference. APSEC, IEEE, pp. 327–337.

Nguyen, S., Manh, C.T., Tran, K.T., Nguyen, T.M., Nguyen, T.-T., Ngo, K.-T., Vo, H.D., 2023a. ARist: An effective API argument recommendation approach. J. Syst. Softw. 111786.

Nguyen, H.A., Nguyen, T.N., Dig, D., Nguyen, S., Tran, H., Hilton, M., 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 819–830.

Nguyen, S., Nguyen, T.-T., Vu, T.T., Do, T.-D., Ngo, K.-T., Vo, H.D., 2024. Code-centric learning-based just-in-time vulnerability detectio. https://github.com/ttrangnguyen/CodeJIT.

Nguyen, S., Phan, H., Le, T., Nguyen, T.N., 2020. Suggesting natural method names to check name consistencies. In: 2020 IEEE 42nd International Conference on Software Engineering. IEEE, pp. 1372–1384.

Nguyen, S., Vu, T.-T.Y., Vo, D.-H., 2023b. VFFINDER: A graph-based approach for automated silent vulnerability-fix identification. In: Proceedings of the 15th IEEE International Conference on Knowledge and Systems Engineering. URL https://arxiv.org/abs/2309.01971.

Ni, C., Wang, W., Yang, K., Xia, X., Liu, K., Lo, D., 2022. The best of both worlds: Integrating semantic features with expert features for defect prediction and localization. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 672–683.

Nie, L.Y., Gao, C., Zhong, Z., Lam, W., Liu, Y., Xu, Z., 2021. Coregen: Contextualized code representation learning for commit message generation. Neurocomputing 459, 97–107.

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

Pornprasit, C., Tantithamthavorn, C.K., 2021. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 369–379.

Purushothaman, R., Perry, D.E., 2005. Toward understanding the rhetoric of small source code changes. IEEE Trans. Softw. Eng. 31 (6), 511–526.

Schlichtkrull, M., Kipf, T.N., Bloem, P., Berg, R.v.d., Titov, I., Welling, M., 2018. Modeling relational data with graph convolutional networks. In: European Semantic Web Conference. Springer, pp. 593–607.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005a. When do changes induce fixes? ACM Sigsoft Softw. Eng. Not. 30 (4), 1–5.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005b. When do changes induce fixes? ACM Sigsoft Softw. Eng. Not. 30 (4), 1–5.

SonarQube, 2022. SonarQube. https://www.sonarsource.com/products/sonarqube/. (Online; accessed 19 October 2022).

Sparks, S., Embleton, S., Cunningham, R., Zou, C., 2007. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual Computer Security Applications Conference. ACSAC 2007, IEEE, pp. 477–486.

Tian, Y., Zhang, Y., Stol, K.-J., Jiang, L., Liu, H., 2022. What makes a good commit message? In: Proceedings of the 44th International Conference on Software Engineering. pp. 2389–2401.

Vo, H.D., Nguyen, S., 2023. Can an old fashioned feature extraction and a light-weight model improve vulnerability type identification performance? Inf. Softw. Technol. 107304.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 397–407.

Wang, H., Xia, X., Lo, D., He, Q., Wang, X., Grundy, J., 2021. Context-aware retrieval-based deep commit message generation. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (4), 1–30.

Wu, B., Liu, S., Feng, R., Xie, X., Siow, J., Lin, S.-W., 2022. Enhancing security patch identification by capturing structures in commits. IEEE Trans. Dependable Secure Comput..

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Yang, L., Li, X., Yu, Y., 2017. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In: GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, pp. 1–7.

Zeng, Z., Zhang, Y., Zhang, H., Zhang, L., 2021. Deep just-in-time defect prediction: How far are we? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 427–438.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems, vol. 32.

Zou, D., Wang, S., Xu, S., Li, Z., Jin, H., 2019. μ VulDeePecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans. Dependable Secure Comput. 18 (5), 2224–2236.

Son Nguyen is a lecturer at the Faculty of Information Technology, the University of Engineering and Technology, VNU Hanoi. He earned a Ph.D. in Computer Science from the University of Texas at Dallas in 2022. He received the B.Sc. degree in Computer Science from Vietnam National University, Hanoi, in 2015. His research interests include program analysis, configurable code analysis, and statistical approaches in software engineering.

E-mail: sonnguyen@vnu.edu.vn

Thu-Trang Nguyen is a Ph.D. candidate in Software Engineering at University of Engineering and Technology, Vietnam National University, Hanoi (VNU - UET). She received her M.Sc. degree from Japan Advanced Institute of Science and Technology in 2019 and received her B.Sc. degree from VNU - UET in 2016. Her research interests include program analysis, software product line systems, software vulnerability detection, and software testing.

E-mail: trang.nguyen@vnu.edu.vn

Thanh Trong Vu earned a Bachelor's degree from VNU University of Technology and Engineering, Vietnam, in 2023. After that, he continues to work as a full-time researcher at the Faculty of Information Technology at the same college. His research interests include source code analysis and machine learning for code.

E-mail: thanhvu@vnu.edu.vn

Thanh-Dat Do is an Undergraduate Student at VNU University of Technology and Engineering, Vietnam. His research interests include source code analysis, smart contracts analysis, and machine learning for code.

E-mail: 20020045@vnu.edu.vn

Kien-Tuan Ngo is a Ph.D. candidate in Computer Science at University of Southern California. He earned the Bachelor's degree from VNU University of Technology and Engineering, Vietnam in 2020. After that, he continues to work as a full-time researcher at the Faculty of Information Technology at the same college. His research interests include source code analysis, software testing, machine learning for code, and empirical software.

E-mail: tuanngokien@vnu.edu.vn

Hieu Dinh Vo is the head of the Department of Software Engineering at the Faculty of Information Technology, the University of Engineering and Technology, VNU Hanoi. He earned a Bachelor's degree in Computer Engineering from the Ho Chi Minh City University of Technology, a Master's degree in Distributed Multimedia Systems from the University of Leeds, and a Ph.D. in Information Science from the Japan Advanced Institute of Science and Technology. His research interests include source code analysis, software testing, service computing, and software architecture.

E-mail: hieuvd@vnu.edu.vn