Journal of Computer Science and Cybernetics, V.40, N.1 (2024), 79-101
DOI no 10.15625/1813-9663/19102

JUST-IN-TIME VULNERABILITY DETECTION AND LOCALIZATION

HIEU DINH VO

Faculty of Information Technology, VNU University of Engineering and Technology, Ha Noi, Viet Nam

Abstract. Software vulnerabilities have increased dramatically, and multiple severe attacks have occurred in recent years. This poses a critical challenge for early detection and prevention of vulnerabilities in Software Quality Assurance. This paper introduces a novel framework, JULY, which serves the dual purpose of detecting vulnerable commits and localizing the root causes of the vulnerabilities. The fundamental concept of JULY is that the determinant of the vulnerability of a commit is the inherent meaning embedded in its changed code. For just-in-time vulnerability detection (JIT-VD), JULY represents each commit by a Code Transformation Graph and employs a Graph Neural Network model to capture their meanings and distinguish between vulnerable and non-vulnerable commits. Once a commit is detected as vulnerable, it is passed to the just-in-time vulnerability localization (JIT-VL) model to localize the root causes, which are vulnerable changed statements. In JIT-VL, JULY encodes each statement by the following features: *operation*, *context*, and *topic*. Then, JULY measures the suspiciousness score of each changed statement and ranks them based on their scores. To evaluate the effectiveness of JULY, we conducted several experiments using a dataset consisting of 20,274 commits in 506 C/C++ projects. JULY achieves a remarkable improvement of 95% in Top-1 ACC and 63% in MRR compared to the state-of-the-art approaches. Furthermore, when examining the same portion (i.e., 20%) of modified statements in each commit, JULY can find twice as many vulnerable statements within a given commit as the state-of-the-art approaches.

Keywords. Just-in-time vulnerability detection; Just-in-time vulnerability localization; Vulnerable commit; Vulnerable statement.

1. INTRODUCTION

Software is an integral component in a massive number of real-world systems. Thus, it is essential to guarantee software robustness and security [3, 14], especially for highly critical systems such as traffic control, aviation coordination, chemical/nuclear industrial operations, etc. Moreover, the number of vulnerabilities has expanded rapidly, from 5,297 in 2012 to 25,082 in 2022 [1]. Therefore, it poses a critical challenge to modern Software Quality Assurance (SQA) practices for accurately detecting and preventing vulnerabilities early in the software development life cycle.

In practice, multiple just-in-time vulnerability/defect detection (JIT-VD) methods [21, 22, 23, 33] have been introduced for early identifying software weaknesses and preventing

*Corresponding author.
E-mail addresses: hieuvd@vnu.edu.vn

them from being merged into source code. These JIT-VD methods can provide prompt feedback for the authors of code commits on their modifications. This can help them quickly fix the vulnerabilities and improve the code quality while the context of their changes remains fresh in their minds. Additionally, these JIT-VD techniques can also assist code auditors in the process of reviewing code commits.

However, most of these methods only focus on the JIT-VD phase [20, 22, 33] and mainly rely on commit messages and expert features such as the number of modified code lines for assessing the suspiciousness of the commits. Indeed, these features may exhibit some connections, but do not inherently imply the existence of vulnerabilities in the code changes. For instance, a commit with a message containing keywords indicative of bug-fixing, such as "fix", "failures", or "resolve" may be categorized as safe. However, a bug-fixing commit could still introduce a new bug [20]. Additionally, the approaches leveraging expert features often predict the code change with multiple added lines as vulnerable since the more complex a commit, the more dangerous it is. Consequently, commits with a few code additions might be incorrectly identified as safe/non-vulnerable. Indeed, commit messages and expert features can be related, yet they do not necessarily lead to the presence of vulnerabilities, which are caused by the semantics of the modifications.

Furthermore, even if a vulnerable commit is correctly detected, it is still wearisome and laborious to manually investigate the whole commit to figure out the vulnerable statements. The reason is that a commit is often tangled [28] and contains a large number of modifications (e.g., about 100 added statements in each commit on average [30]). There are a few approaches to localizing just-in-time vulnerable statements. However, these approaches only consider the lexical features. Therefore, their results are still limited. For example, JITLine [23] localizes vulnerable statements by using LIME [25] to identify the tokens in the commits that most impact the detection results of their model. In addition, JIT-DIL leverages a n-gram language model to measure the suspiciousness of a statement in a commit. However, both of these approaches capture only lexical-level features, but the semantics of tokens and the statements are not considered.

In this paper, we introduce a novel framework, July, which serves the dual purpose of detecting vulnerable commits and fine-grained localizing the root causes of the vulnerabilities in each detected commit. The fundamental concept of July is that the main determinant of the vulnerability of a commit is the inherent meaning embedded in its changed code. Furthermore, we are aware that the just-in-time vulnerability detection (JIT-VD) task and the just-in-time vulnerability localization (JIT-VL) task possess distinct characteristics. To optimize their performance, we design specialized models tailored to the unique demands of each task.

For the JIT-VD, July represents each commit by a Code Transformation Graph (CTG) and subsequently employs a Graph Neural Network (GNN) model to capture their meanings and distinguish between vulnerable and non-vulnerable commits. Once a commit is suspiciously vulnerable, it is passed to the JIT-VL model to identify the suspicious changed statements.

To comprehensively capture the statements' meaning for an effective JIT-VL process, July encodes each statement by the following features: *operation*, *context*, and *topic*. Particularly, the statement's *operation* expresses what the statement is. The *context* captures how the statement works. Since the analyzing statement is a changed statement in a commit, to properly capture the meaning of code change, we consider the *context* in both the code aspect ($context_c$) and the change operators of its elements ($context_o$). In addition, the *topic*, i.e., the name of the containing function, indicates where the statement is. Finally, JULY ranks the changed statements in a suspicious commit based on these statements' likelihood to be vulnerable.

To evaluate the effectiveness of JULY, we conducted several experiments using a dataset consisting of 20,274 commits in 506 C/C++ projects. There are 11,299 non-vulnerable commits, while the remaining 8,975 commits are vulnerable. Among the vulnerable commits, there are approximately 40,000 vulnerable statements and over 1.0 million non-vulnerable statements. The experimental results show that JULY significantly outperforms the existing state-of-the-art approaches in the integrated end-to-end process of JIT-VD and JIT-VL. Specifically, JULY achieves a remarkable improvement of 95% in Top-1 accuracy and an increase of 63% in Mean Reciprocal Rank (MRR). Furthermore, when examining the same portion (i.e., 20%) of modified statements in each commit, JULY can find twice as many vulnerable statements within a given commit as the other approaches. For the stand-alone JIT-VL phase, JULY's performance is significantly better than those of the state-of-the-art approaches by 100%–30% in Top-1 accuracy and 60%-167% in Recall@20%Effort. This demonstrates that by using JULY, we can find more vulnerabilities with less effort compared to the other approaches.

In brief, this paper makes the following contributions:

- Introduce JULY, a novel framework for both JIT-VD and JIT-VL.
- An extensive empirical evaluation showing that JULY significantly outperforms the state-of-the-art approaches.

2. RELATED WORK

Vulnerability detection. Various deep learning-based techniques [4, 5, 6, 9, 10, 11, 16, 17, 26] have been proposed for detecting vulnerabilities in programs at different levels such as components, files, or functions, etc. In these approaches, two types of representation are often utilized: token-based and graph-based. In token-based models, code is considered as a sequence of tokens [16, 17, 26], whereas in graph-based models, code is depicted as a graph [4, 5, 6, 15]. For instance, IVDetect [15], Devign [34], or ReVeal [5] are graph-based vulnerability detection approaches which represent a function as a dependence graph, then employ GNN models for capturing the hidden features and determining whether the function is vulnerable or not. JULY also leverages a GNN model for capturing the code semantics and detecting vulnerabilities. However, different from these methods, which focus on identifying vulnerabilities at the release time, JULY focuses on detecting vulnerabilities at the commit level. In JULY, not only the code elements and their dependence relationships in the graphs but also the change operators are important for determining whether a code change is dangerous or safe. JULY and the release-time vulnerability detection methods can complement each other, providing robust support to developers in ensuring software quality throughout the development process.

Furthermore, JULY aligns with the just-in-time defect/vulnerability detection research [2, 21, 22, 23, 31, 33]. In VCCFinder [22], VulDigger [31], and LAPredict [33], expert features are leveraged, and vulnerable/non-vulnerable commits are classified by machine learning models such as SVMs or Random Forests. JITLine [23] utilizes the expert features and token features using bag-of-words from commit messages and changed code to build a defect prediction model with a random forest classifier. JITFine [21] combines expert features with semantic features extracted by CodeBERT [8] from modified code and commit messages to identify vulnerabilities in the commits. Different from these approaches, our work focuses on capturing the semantics of the changed code for detecting and localizing vulnerabilities at the commit level. In July, the vulnerability of a commit is measured regardless of the commit message but based on only the semantics of code change represented by the code elements, the change operators, and the program dependence relationships.

Vulnerability localization. There are several studies [6, 9, 11, 21, 23, 24, 30] focusing on detecting defects/vulnerabilities at the statement level. For example, LineVD [11], LineVul [9], and Velvet [6] are the state-of-the-art approaches that detect vulnerabilities at the most fine-grained, statement level, or line-level. However, the objective of these methods is to detect vulnerable statements at the release time, which is different from July. July aims to identify changed statements that are dangerous to source code to prevent them from being merged into the project's source.

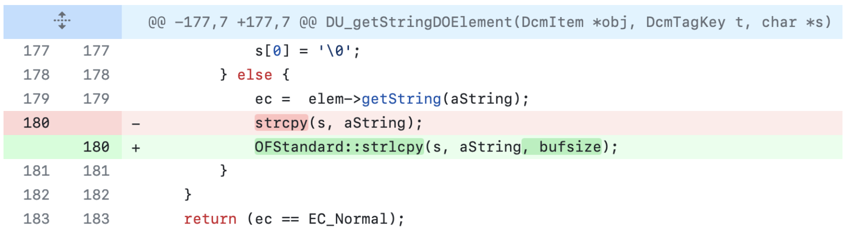
For just-in-time defect/vulnerability localization, JITFine [21] and JITLine [23] leverage information from the detection phase, while JIT-DIL and DeepDL build separated localization models. Specifically, JITFine [21] takes the attention weights of the code tokens to identify the most dangerous tokens. JITLine [23] employs LIME [25] to interpret their models for localization tasks. Both JIT-DIL [30] and DeepDL [24] take the ideas of "naturalness" of source code for localizing just-in-time vulnerabilities. The more unnatural a modified code line, the more suspicious it is. In July, we also design a specialized model for the JIT-VL task. However, different from the previous work, which only captures the lexical level features, we comprehensively capture the semantics of the statements by four features: *operation*, $context_c$, $context_o$, and *topic*. Through these contexts, July can understand code statements and distinguish them to find the root cause of vulnerabilities. This helps July obtain better performance compared to the existing approaches which are demonstrated in the experimental results.

3. MOTIVATION AND GUIDING PRINCIPLES

3.1. Motivating example

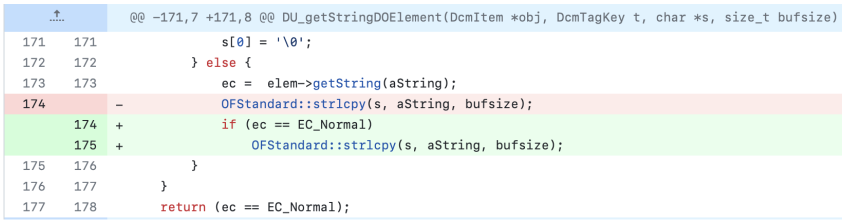
Figure 1a shows a vulnerability in project `DCMTK`[1], which is introduced by commit `e4f7026` on May 11th, 2018. In this commit, for copying a string, this commit replaced the function `strcpy` with a new function `OFStandard:strlcpy`. However, if the source string is `null`, there will not exist any value for copying. It could lead to undefined behaviors. Thus, invoking `OFStandard:strlcpy` without checking the value of `aString` could cause a null pointer deference problem. Until September 15th, 2021 (more than three years after the commit `e4f7026`), this vulnerability was identified and fixed by the commit `5c14bf5` shown in Figure 1b. For existence in such a long period, this vulnerability could be exploited and cause serious problems.

[1]https://github.com/DCMTK/dcmtk



```
@@ -177,7 +177,7 @@ DU_getStringDOElement(DcmItem *obj, DcmTagKey t, char *s)
              s[0] = '\0';
          } else {
              ec =  elem->getString(aString);
-             strcpy(s, aString);
+             OFStandard::strlcpy(s, aString, bufsize);
          }
      }
      return (ec == EC_Normal);
```

(a) A vulnerability introduced by commit `e4f7026`



```
@@ -171,7 +171,8 @@ DU_getStringDOElement(DcmItem *obj, DcmTagKey t, char *s, size_t bufsize)
              s[0] = '\0';
          } else {
              ec =  elem->getString(aString);
-             OFStandard::strlcpy(s, aString, bufsize);
+             if (ec == EC_Normal)
+                 OFStandard::strlcpy(s, aString, bufsize);
          }
      }
      return (ec == EC_Normal);
```

(b) The patch of the vulnerability shown in Figure 1a

Figure 1: Example of vulnerable commit and its corresponding patch in project `DCMTK`

Moreover, this vulnerability introducing commit changed 32 files with 403 additions and 347 deletions.

Consequently, even once this vulnerable commit is detected, manually investigating these changed files to identify the vulnerable statement(s) in this commit is still time-consuming and labor-intensive. Thus, for productive security inspection, it is necessary to assist developers in both detecting vulnerable commits before it is merged into source code (JIT-VD) as well as localizing such vulnerabilities at fined-grain such as statement level (JIT-VL).

3.2. Guiding principles for JIT-VD and JIT-VL

This section introduces the principles guiding our approach for JIT-VD and JIT-VL.

3.2.1. Detecting just-in-time vulnerabilities

Principle 1. *For a commit, the semantics of the code change is the crucial factor for determining its vulnerabilities.*

In practice, one could leverage multiple characteristics of a commit to determine whether it is vulnerable. For example, based on the hypothesis that the more complex a commit, the more suspicious it is, several approaches [22, 30, 31, 33] identify a dangerous commit based on the manually defined expert features such as the number of added lines, etc. Moreover, several other methods [20, 21, 23] capture the meaning of code change and/or the corresponding commit message to measure the suspiciousness of the commit. Meanwhile, the existing approach [20] shows that the approaches considering the code's meaning obtained better detection performance. This demonstrates that code change is essential for determining a commit's danger.

Indeed, expert features or commit messages could be useful guidance for examining a commit. However, a commit is dangerous to the source code if it introduces vulnerabilities to the program via the changed lines.

For example, the commit in Figure 1a is considered a dangerous commit because the added code line does not check null before copying the value of the parameter `aString`, regardless of the number of added lines in this commit or the commit message. Thus, to precisely determine whether a commit is vulnerable, the meaning of the code change is the key factor that should be represented and investigated appropriately.

Principle 2. *The relations of the changed statements with the others, including both changed and unchanged statements, are important for capturing the meaning of the code change.*

The program's behavior is formed from the code statements and their interactions via control/data dependencies.

In a commit, changed statements are introduced to create new behaviors or modify old behaviors of the program. Thus, the dependencies among changed statements must be analyzed to interpret the newly introduced behaviors and how they are constructed. In addition, to recognize and explain which old behaviors are modified and how they are affected, the relations of the changed statements with the other unchanged statements also need to be investigated.

In summary, as the change code meaning is the primary determinant to effectively detect just-in-time vulnerability, code change should be well represented for capturing both the changed and semantically related unchanged statements and their relations.

3.2.2. Localizing just-in-time vulnerabilities

Principle 3. *To precisely assess the suspiciousness of a statement, it is essential to understand the operation of the statement.*

Indeed, each statement is responsible for a specific *operation* that contributes to the purpose of the commit and eventually contributes to the whole behaviors of the function/program. To precisely localize the most suspicious statements in the commit, which are likely to be the root cause of the vulnerability, the suspiciousness score of each statement must be accurately assessed. Thus, capturing the meaning of each statement, e.g., what the statement is or what it does, is essential. For example, we cannot conclude whether s_{180} in Figure 1a is vulnerable if we do not know the functionality of `OFStandard::strcpy`, i.e., what the function does and what the corresponding parameters are. Therefore, analyzing the statement's *operation* is of first importance for understanding the statement.

Principle 4. *Assessing a statement's suspiciousness should consider the statement in its context.*

In a program, a statement does not solely execute to complete its operation and contribute to the behaviors of the program. Instead, the statement interacts with the others regarding control and data dependencies (the statement's *context*). Thus, to precisely understand the statement, not only the statement itself but also its contexts should be considered. For instance, the vulnerability introduced by line 180 in Figure 1a is caused by the null value of variable `aString`. With only the statement s_{180}, we cannot know whether `aString` can have a null value or whether it is carefully checked before copying. This information can be obtained by the statements having control/data dependencies with s_{180}. In addition, statements s_{180} in Figure 1a and s_{175} in Figure 1b are identical, but the former statement is vulnerable, and the latter statement is safe. The reason is that statement s_{174} in Figure. 1b

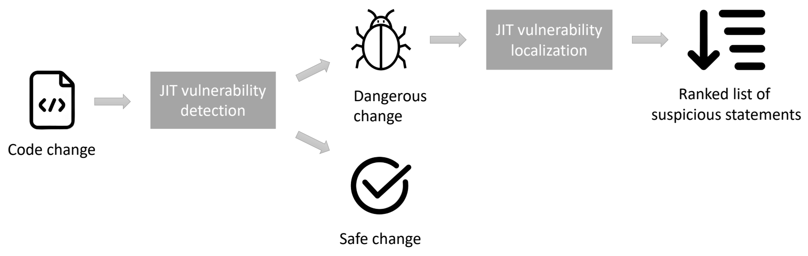

Code change
JIT vulnerability detection
Dangerous change
Safe change
JIT vulnerability localization
Ranked list of suspicious statements

Figure 2: JULY framework's overview

guarantees that s_{175} can copy the value of variable `aString` only if it successfully gets a non-null value. As a result, the execution of s_{175} in Figure 1b is safe.

Furthermore, to capture the context of a changed statement s in a commit, besides the code aspect of the context, the change operators of the contexts are also valuable. Specifically, the code aspect of the context ($context_c$) of s is the code statements that impact/be impacted by s. These statements have an impact on the behaviors of s. Thus, their importance in understanding s is apparent. In addition, the change operators of the context ($context_o$) of s specify how each code element in the context is added/deleted/unchanged. Under the circumstance of analyzing a vulnerable commit, $context_o$ helps to understand how the source code is transformed from the old version to the new one and then leads to a vulnerability. Therefore, both the code aspect and change operators of the context, $context_c$, and $context_o$, are essential when analyzing the context.

Principle 5. *The topic feature, such as the function name, could be valuable for narrowing down the search space.*

A vulnerable commit often contains many changed lines (e.g., about 100 added statements on average [30]) across multiple files and functions. Taking the (general) *topic* feature, such as the name of the containing function, could help the model focus on the vulnerable-prone function instead of considering the whole commit equally. Thus, the *topic* feature could help to narrow down the search space for better localizing specific vulnerable statements.

In summary, to precisely localize the just-in-time vulnerable statements, the changed statements' operations and their contexts (including both $context_c$ and $context_o$) need to be appropriately represented. In addition, the topic feature could be helpful in narrowing down the search space.

4. JUST-IN-TIME VULNERABILITY DETECTION AND LOCALIZATION FRAMEWORK

Figure 2 shows the overview of our proposed JIT-VD and JIT-VL framework, JULY. For a commit, JULY's JIT-VD model accesses and identifies whether that commit is vulnerable or not. Next, if it is detected as a vulnerable commit, the JIT-VL model further analyzes to measure the suspiciousness score of each statement in the commit and returns a ranked list of suspicious statements.

4.1. Code change representation

Following the guiding principles discussed in Section 3.2., we leverage the Code Transformation Graph (CTG) [20] to capture the changed code and its relations with the other semantically related parts in the program. Specifically, a CTG is an annotated graph representing the changed/unchanged code elements of the program before and after the change is applied, as well as the relations among these code elements. Formally, the CTG is defined as follows.

Definition 1. (Code transformation graph (CTG) [20]) For a commit changing code from a version v_o to another version v_n, the *code transformation graph*, $\mathcal{G} = \langle \mathcal{N}, \mathcal{E}, \mathcal{R}, \alpha \rangle$ representing the code elements and their relations in both versions respectively, in which:

- $\mathcal{N}$ consists of the code elements (i.e., AST nodes) in both v_o and v_n.
- $\mathcal{E}$ is the set of edges representing the relations between nodes, For $n_i, n_j \in \mathcal{N}$, an edge exists from n_i to n_j regarding relation $r \in \mathcal{R} = \{\textit{structure, dependency}\}$, $\exists e^r_{ij} = \langle n_i, r, n_j \rangle \in \mathcal{E}$ if there is a relation r between n_i and n_j in a code version.
- $\mathcal{R}$ is a set of the considered relations between code elements, $\mathcal{R} = \{\textit{structure, dependency}\}$.
- Annotations for nodes and edges are either *unchanged*, *added*, or *deleted* by the change. Formally, $\alpha(g) \in \{\textit{unchanged, added, deleted}\}$, where g is a node in $\mathcal{N}$ or an edge in $\mathcal{E}$:
  - + $\alpha(g) = \textit{added}$ if g is a node or edge which contained in v_n and not contained in v_o.
  - + $\alpha(g) = \textit{deleted}$ if g is a node or edge which contained in v_o and not contained in v_n.
  - + Otherwise, $\alpha(g) = \textit{unchanged}$.

Figure 3 shows a partial CTG of the commit in Figure 1a. In statement s_{180}, the called function `strcpy` is deleted, and the new function `OFStandard::strlcpy` is added. Also, one more parameter `bufsize` is added for this function. In addition, this graph also demonstrates the relation of this changed statement, s_{180}, with the others. For instance, s_{180} is control dependent on s_{176} and data-dependent on s_{179}.

4.2. Just-in-time vulnerability detection model

In the detection phase, JULY employs a Relational Graph Convolution Network (RGCN) [27] to capture the semantics of code changes represented by the nodes and their relations in CTGs. Next, a Multilayer Perceptron (MLP) is used to learn the patterns and classify vulnerable and non-vulnerable commits. The overview of the JIT-VD model is demonstrated in Figure 4.

First, we embedded the nodes in CTGs into numeric vectors before feeding them into an RGCN model. In this work, the feature vector of each node in CTG contains information about the node content and its annotated changed operator (*added*, *deleted*, or *unchanged*). For embedding node content into a d-dimensional vector c_i, we employ Word2vec [19], which is widely used for embedding semantics of code tokens [7]. The changed operator is embedded by a one-hot vector, o_i. Then, the feature vector of node i is the concatenation of its node content vector and changed operator vector, $h_i = c_i \bigoplus o_i$.

The embed CTGs, whose nodes are encoded appropriately, are fed to an RGCN to learn their important features. Each layer of RGCN computes the representations for the nodes

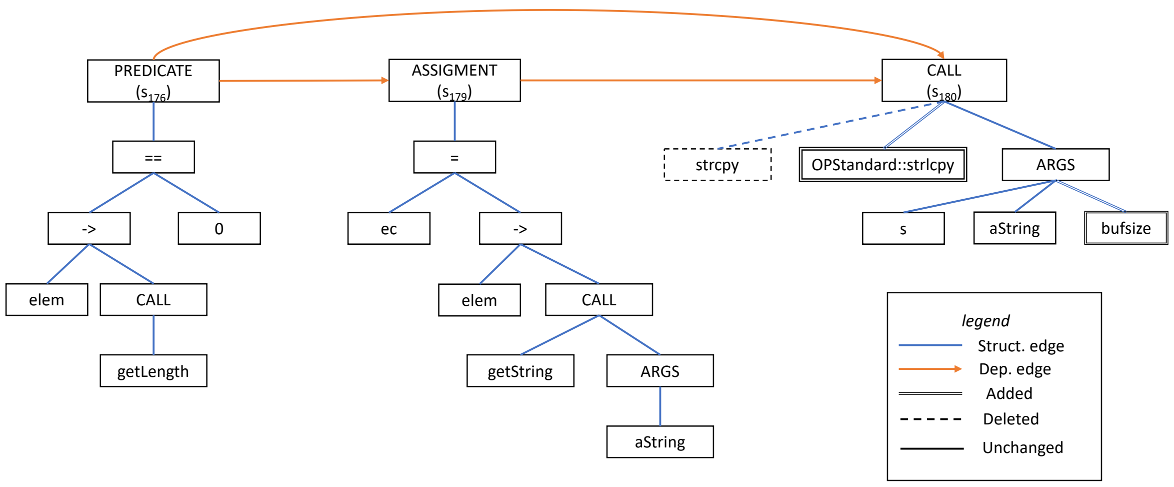


Figure 3: A partial CTG of the commit shown in Figure 1a

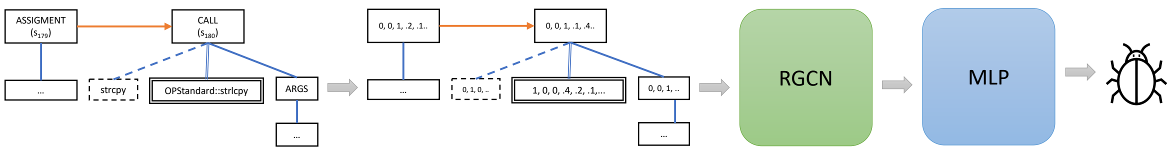


Figure 4: Just-in-time vulnerability detection model

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.

After L RGCN layers, a k-dimensional graph-level vector representation 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. Finally, the graph features are then passed to an MLP to classify if $\mathcal{G}$ is vulnerable or not.

4.3. Just-in-time vulnerability localization model

July's JIT-VL model is shown in Figure 5. For each statement, July extracts the corresponding features, including *operation*, $context_c$, $context_o$, and *topic*. Then, these features are embedded appropriately. After that, these embedding vectors are concatenated to represent the whole feature map of the statement. The final vector is fed into fully connected layers. The output of the last layer is the predicted suspiciousness score of the statement.

4.3.1. Representation learning

This section presents how we build a representation vector for statements in a detected vulnerable commit. For each statement s, we extract and represent it by four features: *operation*, $context_c$, $context_o$, and *topic*.

Operation is the specific task which s is responsible for in the function/program. In Figure 1a, statement s_{179} gets a value type `String` from object `elem`, and assigns the returned value into `ec`. Statement s_{180} performs copying a value of variable `aString` with size `bufsize` into variable `s`.

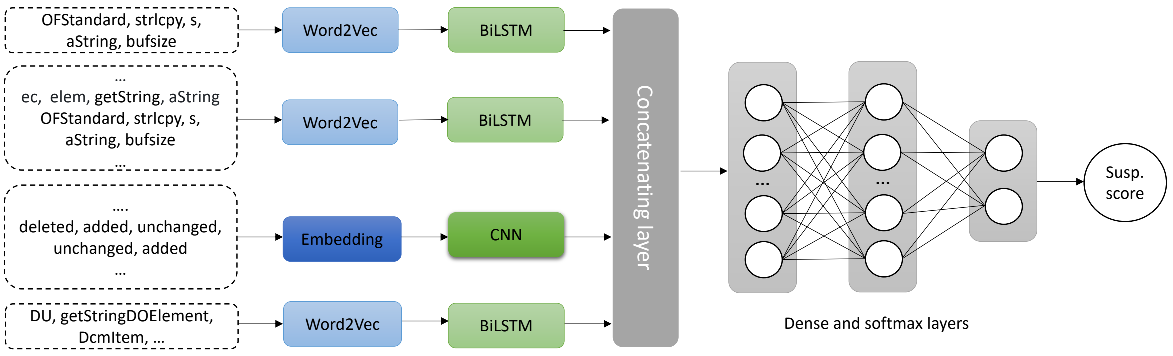

OFStandard, strlcpy, s, aString, bufsize
...
ec, elem, getString, aString
OFStandard, strlcpy, s, aString, bufsize
...
....
deleted, added, unchanged, unchanged, added
...
DU, getStringDOElement, DcmItem, ...
Word2Vec
Word2Vec
Embedding
Word2Vec
BiLSTM
BiLSTM
CNN
BiLSTM
Concatenating layer
Dense and softmax layers
Susp. score

Figure 5: Just-in-time vulnerability localization model

To capture such *operation* of s, we aim to embed the code tokens constructing s. Specifically, we apply lexical analysis to tokenize s into a sequence of tokens. Next, the Word2vec [19] model is employed for embedding each token into a numeric vector. After that, to capture the relationship of tokens in the whole statement, the vectors of these tokens are fed into a BiLSTM network. BiLSTM is applied because of the following reasons. First, code tokens must appear together in a certain order to make the program syntactically correct [18]. In addition, code tokens in a statement often have a particular relation with their preceding and succeeding tokens [12].

$Context_c$ contains the statements in the CTG of the suspicious commit which are semantically related to s. The context represents the situation where s belongs. This feature helps to position the task of s in a particular context and thus helps to understand s precisely.

Moreover, it is unreasonable to conclude that s is vulnerable regarding its context. For example, although it is clear that s_{180} is vulnerability-prone since this statement invokes a function that can lead to undefined behaviors, we cannot conclude it is a vulnerable statement without considering its control/data dependence. This statement is vulnerable in the context where its second variable, `aString`, is not guaranteed to be non-null before invoking this function. Thus, considering the statements having control/data dependence with s is necessary to assess its suspiciousness.

JULY extracts the $context_c$ of s by conducting both backward and forward slicing in the CTG of the commit, starting from s. For example, the $context_c$ of statement s_{180} in Figure 1a containing s_{176}, s_{179}, etc. For embedding the $context_c$, we also tokenize the statements in the $context_c$ into a sequence of tokens and then vectorize by them Word2vec and BiLSTM models.

$\underline{Context_o}$ is the sequence of annotated change operators (*added/deleted/unchanged*) of code elements in the context of s. Indeed, besides the semantics of the modified statements ($context_c$), how the statements are changed, such as added or deleted, also helps to capture the changes' meaning.

Moreover, capturing a general pattern in the change operator sequence ($context_o$) is easier than capturing the semantics since its information is looser than the information contained in the $context_c$. Therefore, this feature can provide valuable guidance and can be combined with the semantics of the $context_c$ to better evaluate the suspiciousness of s.

In this framework, we use the Ordinal Encode technique for embedding the changing pattern of the $context_o$ of s. The reason is that the values of changed operators are categorical, and the number of operators is limited. Next, we fed the encoded vector into a CNN network to obtain the whole change pattern of the *context*$_o$.

Topic is a global/general feature that could help to narrow down the search space by identifying whether s is contained in a vulnerable function. In this work, we take the function name as the *topic* of s. Similar to *operation* and *context*$_c$, the *topic* is also lexical tokenized and then vectorized by Word2vec and BiLSTM models.

4.3.2. Suspiciousness assessment

To accumulate each feature by keeping their most important elements and also reduce the output dimension, JULY employs a *Global Max Pooling* layer following each BiLSTM/CNN network. After that, to obtain the patterns of the statement, the representation vectors of the four features are concatenated into a unified one to represent a whole feature map of s. The unified vectors are then fed into several fully connected layers. The last layer is activated by a *Softmax* function and has two hidden units. The value of this layer, which corresponds to the probability of being a vulnerable statement of s, is considered as the suspiciousness score of s. A higher score implies a greater probability of s being a vulnerable statement.

4.3.3. Data imbalance handling

Class imbalance is an inevitable challenge of vulnerability detection problems in general. This imbalance of labels could negatively impact the deep learning models' performance since they are often biased by the majority class [5, 32]. In our dataset, the ratio of vulnerable and non-vulnerable statements is 1:27, which is severely imbalanced. To mitigate the impact of this extreme imbalance, we aim to balance the training data set by the under-sampling technique, which has been demonstrated to be the best choice if recall is pursued [32]. Moreover, the impact of the data balancing techniques is also empirically investigated in Subsection 6.5.1.

5. EXPERIMENTAL METHODOLOGY

5.1. Research questions

For evaluation, we seek to answer the following research questions. First, we follow the same procedure in the existing work [21, 23, 30] to evaluate the performance of JULY and compare JULY with the state-of-the-art end-to-end JIT-VD and JIT-VL approaches:

RQ1: End-to-End Performance Evaluation & Comparison. *How effective is our end-to-end JIT-VD and JIT-VL framework compared with the state-of-the-art approaches?*

Additionally, we evaluate the performance of JULY as a stand-alone JIT-VL approach:

RQ2: Stand-alone Performance Evaluation & Comparison. *How effective is our JIT-VL phase's performance compared with the state-of-the-art baselines?*

We also applied the same procedure in the existing studies [13, 24, 30, 33] to evaluate JULY's performance in learning from a set of projects and testing for another set of projects (cross-project):

RQ3: Cross-project Evaluation. *How effective is* JULY *in the cross-project setting?*

Table 1: Dataset statistics [20]

	#Non-vul. commits	#Vul. commits	%adds	$\lvert N\rvert/\lvert E\rvert$ 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

Furthermore, we perform several experiments to evaluate the impacts of different aspects on July's JIT-VL performance and running time:

RQ4: Feature Analysis. *How do different features impact* July*'s performance?*

RQ5: Other Factors. *How do different factors affect* July *'s performance, such as imbalanced training data and commit length?*

RQ6: Time Complexity. *What is the time complexity of* July*?*

5.2. Dataset

To evaluate our approach and compare the performance with the baselines, we employ the benchmark proposed by Nguyen et al. [20]. To the best of our knowledge, this is the newest and largest dataset of vulnerable and non-vulnerable commits. This dataset contains 20,274 commits from 506 C/C++ projects. There are 11,299 non-vulnerable commits and 8,975 vulnerable commits. Among the vulnerable commits, there are approximately 40,000 vulnerable statements and over 1.0 million non-vulnerable statements.

Table 1 shows the statistics of the dataset provided by Nguyen et al. [20]. The table shows the detailed number of commits, the average percentage of added statements in each commit, and the average ratio of the numbers of nodes and edges in each CTG by each project. Specifically, most of the vulnerable and non-vulnerable commits are collected from popular C/C++ projects such as FFmpeg, Qemu, and Linux. The average percentage of added statements is 73%, and the average ratio of nodes and edges in each CTG is 0.7.

5.3. Experimental procedure and evaluation metrics

5.3.1. Experimental procedure

RQ1: End-to-End Performance Evaluation & Comparison.

Baselines: We compare the performance of July with the state-of-the-art end-to-end JIT-VD and JIT-VL approaches, including JITFine [21], JITLine [23], and JIT-DIL [30].

- JITFine [21]: Deep learning-based approach using CodeBERT to embed changed code and commit message features, and then combine with expert features to detect buggy commits. Next, to localize the buggy statements, it leverages the weight of each token obtained from the attention mechanism in CodeBERT.

- JITLine [23]: This tool utilizes changed code and expert features to detect buggy commits and apply LIME [25] for localizing buggy statements.
- JIT-DIL [30]: Expert features are used to detect buggy commits, and a language model, i.e., n-gram, is used to localize buggy statements.

Procedure: We evaluate the performance of the approaches in the real-world time-aware setting as in the related studies [2, 20, 21, 22, 23, 33]. We divided the commits into those before and after time point t. We selected a time point t to achieve a training/test split ratio of 80/20 based on time. The commits before t were used for training, while the commits after t were used for testing. The detail of data splitting is shown in Table 2.

Table 2: Data splitting by time point t for end-to-end performance evaluation

	Commit		Statements	
	#Vul. commits	#Non-vul commits	#Vul. stmts	#Non-vul. stmts
Training	7,748	8,471	33,844	907,050
Testing	1,227	2,828	6,077	168,780

Table 3: Data splitting for the cross-project evaluation

	Commit		Statements	
	#Vul. commits	#Non-vul commits	#Vul. stmts	#Non-vul. stmts
Training	7,714	9,176	34,645	864,075
Testing	1,261	2,123	5,276	211,755

RQ2: Stand-alone Performance Evaluation & Comparison. In this experiment, we measure the JIT-VL performance of the approaches on the same set of detected vulnerable commits. We select the set of vulnerable commits which are correctly detected by all four approaches, July, JITFine, JITLine, and JIT-DIL. Then, we compare the JIT-VL of the approaches on this set.

RQ3: Cross-project Evaluation. Similar to existing work [13, 24, 30, 33], in this experiment, 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 whole set of projects is randomly split into 80% (402 projects) for training and 20% (104 projects) for testing. The detail of data splitting by projects is shown in Table 3.

RQ4: Feature Analysis. To analyze the impact of each feature on July's localization performance, we build different variants of July by alternatively excluding each feature.

RQ5: Other Factors. We studied the impacts of the following factors on the performance of July: the imbalance of training data and change size. For the impact of data imbalance on July's performance, we build and evaluate different variants of July, which are trained with the original imbalanced training data set and the re-balanced dataset. For studying the impact of change size, we gradually vary the range of the change size and analyze July's results.

5.3.2. Evaluation metrics

Top-k Accuracy measures whether the first k-ranked statements are the vulnerable statements. Given a vulnerable commit c, if at least one vulnerable statement of c is ranked in top-k, we consider an accurate localization, $topk(c) = 1$. Otherwise, we consider an inaccurate localization, $topk(c) = 0$. For a set of n commits, top-k accuracy is measured as $TopK\_ACC = \frac{1}{n}\sum_{i=1}^{n} topk(c_i)$. In this work, we evaluate $k = [1, 3, 5, 7, 10, 20]$ as the experimental setting in the related studies [21, 23, 24, 30].

MRR measures how far we need to check down a sorted list to find the first vulnerable statement. In other words, *MRR* is the average of the reciprocal ranks for a set of commits. For a commit c, its reciprocal rank is the multiplicative inverse of the rank of the first correctly localized vulnerable statement ($rank_{c_i}$). For a set of n commits, *MRR* is measured as $MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_{c_i}}$.

MAP is the mean of the average precision of all the vulnerable statements in each commit. For a commit c, which has total m statements and v of them are vulnerable statements, the average precision (AP) of c is $AP(c) = \frac{1}{v}\sum_{j}^{m} P(j) \times rel(j)$.

In this equation, the $P(j)$ is the precision at cut-off j in the ranked list of statements of c, e.g., the percentage of vulnerable statements correctly ranked in the first j statements. $rel(j) = 1$ if the statement at rank j is a vulnerable statement, otherwise $rel(j) = 0$. $MAP = \frac{1}{n}\sum_{i}^{n} AP(c_i)$.

Recall@X%Effort measures the proportion of vulnerable statements that can be correctly found with a given $X\%$ effort. In this work, we applied the same procedure in existing work [21] considering $X = 20\%$ modified statements of a given commit.

6. EXPERIMENTAL RESULTS

6.1. RQ1: End-to-End Performance Evaluation & Comparison

Table 4: RQ1. End-to-End Performance Evaluation & Comparison

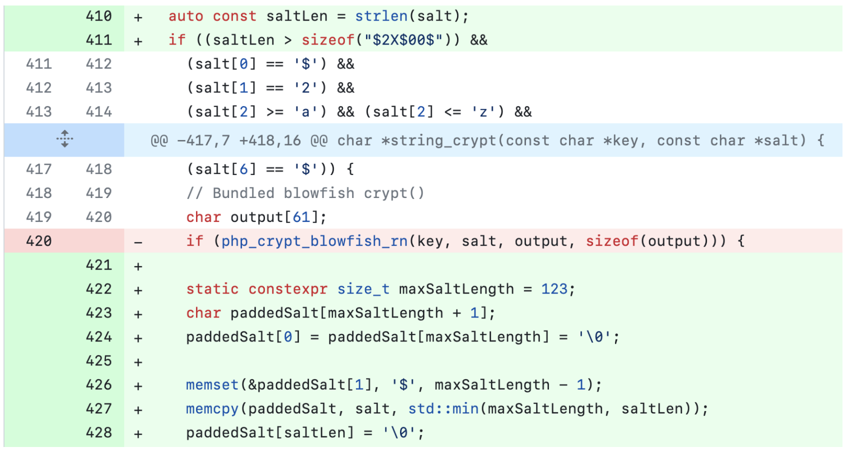
	July	JITFine	JITLine	JIT-DIL
Top-1 ACC	0.19	0.14	0.09	0.08
Top-3 ACC	0.32	0.25	0.18	0.16
Top-5 ACC	0.38	0.31	0.24	0.22
Top-7 ACC	0.43	0.35	0.28	0.26
Top-10 ACC	0.47	0.40	0.34	0.31
Top-20 ACC	0.55	0.49	0.43	0.42
MRR	0.28	0.22	0.16	0.15
MAP	0.23	0.20	0.13	0.13
Recall@20%Effort	0.31	0.17	0.19	0.13

Table 5: RQ2. Stand-alone Performance Evaluation & Comparison

	July	JITFine	JITLine	JIT-DIL
Top-1 ACC	0.12	0.05	0.06	0.03
Top-3 ACC	0.20	0.11	0.12	0.08
Top-5 ACC	0.24	0.15	0.17	0.12
Top-7 ACC	0.28	0.19	0.20	0.15
Top-10 ACC	0.31	0.22	0.23	0.18
Top-20 ACC	0.37	0.29	0.30	0.26
MRR	0.18	0.10	0.11	0.08
MAP	0.14	0.08	0.08	0.06
Recall@20%Effort	0.24	0.13	0.15	0.09

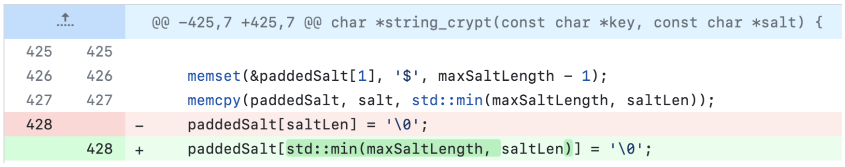
Table 4 shows that July significantly outperforms the other approaches in detecting and localizing just-in-time vulnerabilities.

Particularly, July's results are much better than those of the others, about 95% in Top-1 ACC and 63% in MRR. Especially, the results of July are two times better than JITLine and JIT-DIL. As seen, the Top-1 ACC of July is 19%, while the figures for JITFine, JITLine, and JIT-DIL are only 14%, 9%, and 8%, respectively. In other words, by using July, we can correctly find the vulnerabilities of 19% of the commits right after investigating the first ranked statement of each commit. Meanwhile, JITFine can correctly rank the vulnerable statements first for 14% of the commits. JITLine and JIT-DIL can do that for only 9% and 8% of the commits. To obtain similar top results with July (i.e., the Top-1 ACC), we need to analyze about 2 or 3 statements in each ranked list of JITFine



```
      410  +  auto const saltLen = strlen(salt);
      411  +  if ((saltLen > sizeof("$2X$00$")) &&
411   412        (salt[0] == '$') &&
412   413        (salt[1] == '2') &&
413   414        (salt[2] >= 'a') && (salt[2] <= 'z') &&
           @@ -417,7 +418,16 @@ char *string_crypt(const char *key, const char *salt) {
417   418        (salt[6] == '$')) {
418   419        // Bundled blowfish crypt()
419   420        char output[61];
420        -     if (php_crypt_blowfish_rn(key, salt, output, sizeof(output))) {
      421  +
      422  +     static constexpr size_t maxSaltLength = 123;
      423  +     char paddedSalt[maxSaltLength + 1];
      424  +     paddedSalt[0] = paddedSalt[maxSaltLength] = '\0';
      425  +
      426  +     memset(&paddedSalt[1], '$', maxSaltLength - 1);
      427  +     memcpy(paddedSalt, salt, std::min(maxSaltLength, saltLen));
      428  +     paddedSalt[saltLen] = '\0';
```

(a) Commit abe0b29 introducing a vulnerability at line 428 into source code of project *hhvm*



```
           @@ -425,7 +425,7 @@ char *string_crypt(const char *key, const char *salt) {
425   425
426   426        memset(&paddedSalt[1], '$', maxSaltLength - 1);
427   427        memcpy(paddedSalt, salt, std::min(maxSaltLength, saltLen));
428        -     paddedSalt[saltLen] = '\0';
      428  +     paddedSalt[std::min(maxSaltLength, saltLen)] = '\0';
```

(b) Commit 08193b7 fixing the vulnerability in Figure 6a

Figure 6: Example of vulnerable statements which correctly localized in Top-7 by July

and JITLine, and even five statements in the output of each commit produced by JIT-DIL. These results demonstrate that July can help developers correctly find vulnerable statements in more commits while investigating fewer statements in each commit compared to the state-of-the-art approaches.

For Recall@20%Effort, the results of July are considerably better than the state-of-the-art approaches, up to about 95% relatively. Specifically, July's result is 31%, while JITFine and JITLine obtain 17% and 19%, and JIT-DIL obtains only 13% for this metric. This means that by investigating the same number of statements in each commit (i.e., 20% of changed statements), July can find two times more vulnerable statements in a commit than those of the other approaches.

Interestingly, although the Recalls of the JIT-VD phase of the approaches are slightly different, July still achieves the highest JIT-VL results. Specifically, both July and JIT-DIL correctly detect 70% of vulnerable commits. The JIT-VD performance of JITLine is slightly lower; it can identify 66% of vulnerable commits. Meanwhile, JITFine obtains a slightly higher performance, 73% in Recall. However, the integrated JIT-VD and JIT-VL results of July are much better than those of the existing approaches by 30%–77%. This illustrates that building a specialized model for the specific task of localizing vulnerability helps July comprehensively capture the semantics of the code statements and better localize vulnerabilities at the commit level.

Figure 6a shows an example of a vulnerable statement introduced by commit abe0b29 in project hhvm [2] of Facebook. Specifically, at line 428, they added a character '\0' into

[2]https://github.com/facebook/hhvm

`paddedSalt` to denote the end of this string. However, the allocated space for this variable is `maxSaltLength + 1` (line 423). If `saltLen` is larger than `maxSaltLength`, the string terminator will be set at an invalid index. This could cause undefined behaviors if `paddedSalt` is used. This vulnerability is fixed in commit `08193b7` as shown in Figure 6b.

Indeed, it is very challenging to localize this vulnerability because of two reasons. First, this vulnerability introducing commit is a large commit involving 21 changed files with 185 additions and 44 deletions. Second, to correctly localize this vulnerability, it needs to capture the dependencies of the code statements to understand that the accessing index (`saltLen`) is different from the maximum allocated memory of `paddedSalt`. For this vulnerability, both JITFine and JITLine fail to effectively localize it. Specifically, JITFine localizes this statement at 31^{St}, while JITLine ranks it at the last of the list. In addition, the result of JIT-DIL is better, at 13^{St}, but still out of Top-10. Meanwhile, by capturing the semantics of the analyzing statements and their contexts, July can localize this vulnerable statement at 7^{th}, which is much better than the other approaches.

6.2. RQ2: Stand-alone Performance Evaluation & Comparison

For the same set of detected vulnerable commits, July*'s JIT-VL considerably surpasses the other baselines.* In Top-1 ACC, July's performance is significantly better than those of the others by 100%–300%. Specifically, July's Top-1 ACC is 12%, while the figures of JITLine, JITFine, and JIT-DIL are 6%, 5%, and 3%, respectively. In addition, by investigating the three first-ranked statements of each commit, July can correctly find vulnerable statements of 20% of commits. Meanwhile, by using JITFine, JITLine, or JIT-DIL, we must investigate up to 10 statements in each commit to find the vulnerabilities in the same number of commits.

July *localizes all the vulnerable statements in each commit more precisely than the existing approaches.* Particularly, July's MAP is better than the others by about two times. July's MAP is 0.14, while the MAP of JITFine, JITLine, and JIT-DIL are 0.08, 0.08, and 0.06, respectively. This shows that July can help identify each commit's vulnerable statements much more effectively. Furthermore, using the same effort, i.e., by investing 20% of modified statements in each commit, July can find from 60% to 167% more vulnerable statements in each commit compared to the baselines. July's Recall@20%Effort is 0.24, meanwhile, these figures of JITFine and JITLine are 0.13 and 0.15, respectively. The result of JIT-DIL is even worse; it obtains only 0.09 in Recall@20%Effort.

In July, we designed different models specialized for JIT-VD and JIT-VL tasks, which helped July optimize its performance in each phase. For the JIT-VD phase, July needs to capture the general features of the whole commit. Thus, we employ an RGCN model for learning the representation of the whole CTG. For the JIT-VL phase, it is essential to understand the semantics of each statement in the commit to distinguish them and identify which is the root cause of the vulnerability. Therefore, we explicitly extract related features of each statement and then build different models for appropriately representing these features. This helps July achieve better performance in both end-to-end comparison experiments (Subsection 6.1) and stand-alone comparison experiments. For an example of the vulnerability in Figure 6a, July can localize it better since it considers both the operation of the statement s_{428} and its dependencies such as s_{423} during the JIT-VL process.

Meanwhile, the other approaches that fail to localize the vulnerability in Figure 6a could

Table 6: RQ3. Cross-project evaluation

	July	JITFine	JITLine	JIT-DIL
Top-1 ACC	0.14	0.11	0.10	0.07
Top-3 ACC	0.26	0.21	0.19	0.17
Top-5 ACC	0.33	0.26	0.26	0.24
Top-7 ACC	0.37	0.30	0.29	0.29
Top-10 ACC	0.42	0.35	0.35	0.34
Top-20 ACC	0.53	0.44	0.44	0.46
MRR	0.23	0.18	0.18	0.16
MAP	0.20	0.15	0.14	0.13
Recall@20%Effort	0.20	0.19	0.20	0.15

be due to the following reasons. JITLine and JIT-DIL are ineffective in localizing this vulnerability since they only capture the lexical features. Specifically, JIT-DIL leverages a n-gram language model to measure the suspiciousness of a statement in a commit. However, n-gram can only capture lexical-level features. In addition, JITLine employs LIME to interpret their model's results. However, their model predicts the vulnerability of a commit based on Bag-of-Tokens features (i.e., the frequency of each code token in a commit). Thus, the semantics of tokens and the statements are not captured. Moreover, JITFine localizes the vulnerabilities based on the weights of tokens obtained from the attention mechanism in CodeBERT [8]. However, JITFine considers only modified statements regardless of their contexts. This could lead the model to incorrectly understand the changes, focus on the irrelevant parts, and negatively affect the JIT-VL performance.

6.3. RQ3: Cross-project evaluation

Table 6 shows the performance of detecting and localizing just-in-time vulnerabilities of the approaches in cross-project evaluation experiments. In this experiment, the models are trained from the set projects different from the testing set. *Overall,* July *still obtains much higher performance than the other approaches, which demonstrates that* July *can perform better at learning and generalizing vulnerable patterns.* For instance, the Top-1 ACC of July is higher than the state-of-the-art by about 60%. This result shows that by investing the first ranked statement in the output of each commit, July can find about 60% more vulnerabilities than those found by the others. Moreover, July's MAP is also much better than those of the state-of-the-art approaches by 33%–54%. In particular, July's MAP is 0.20, while the corresponding figures of the JITFine, JITLine, and JIT-DIL are 0.15, 0.14, and 0.13, respectively. This means that to localize all vulnerable statements in a commit, July is also more effective than the other methods.

6.4. RQ4: Feature analysis

Table 7 illustrates how each feature affects July's JIT-VL performance. *In general,* July *obtains the best performance when all the features are employed to represent code statements.* Specifically, if all the features are applied, July's Top-1 ACC is 19%. Meanwhile, if one of the features is excluded, July's result declines by 5%–37%.

Table 7: RQ4 – Impact of features on JULY's performance

	JULY	Not applied feature			
		Operation	*Topic*	*Context$_c$*	*Context$_o$*
Top-1 ACC	0.19	0.17	0.18	0.12	0.18
Top-3 ACC	0.32	0.30	0.30	0.24	0.32
Top-5 ACC	0.38	0.37	0.37	0.29	0.37
Top-7 ACC	0.43	0.41	0.41	0.32	0.42
Top-10 ACC	0.47	0.45	0.46	0.37	0.45
Top-20 ACC	0.55	0.52	0.53	0.45	0.54
MRR	0.28	0.26	0.27	0.20	0.27
MAP	0.23	0.22	0.22	0.17	0.22
Recall@20%Effort	0.31	0.27	0.28	0.20	0.29

Table 8: RQ5 – Impact of the imbalanced labels on JULY's performance

	Original (Imbalanced)	Under-sampling	Over-sampling
Top-1 ACC	0.10	0.19	0.17
Top-3 ACC	0.19	0.32	0.31
Top-5 ACC	0.23	0.38	0.38
Top-7 ACC	0.27	0.43	0.42
Top-10 ACC	0.31	0.47	0.47
Top-20 ACC	0.39	0.55	0.56
MRR	0.17	0.28	0.27
MAP	0.15	0.23	0.23
Recall@20%Effort	0.13	0.31	0.33

Context$_c$ has the highest impact on JULY*'s performance.* If context is excluded, JULY's accuracy significantly drops from 19% to 12% in Top-1 ACC. The reason is that the *context* provides comprehensive information for capturing the semantics of the analyzing statements. For example, the *context* of statement s_{428} in Figure 6a provides information about the allocated space of `paddedSalt` (s_{423} and s_{422}) and also the information about the assessing index `saltLen` (s_{410}). Without such information, it is difficult to confirm the vulnerability of s_{428}.

Operation, topic, and context$_o$ features slightly affect JULY*'s performance.* For instance, Recall@20%Effort of JULY when all the features are enabled is 0.31. If one of these features is alternatively disabled, the results of JULY are 0.27, 0.28, and 0.29, respectively. The reason is that the *topic* and *context$_o$* features are two additional features that could boost the performance of the approach by helping the models focus on important parts of the commits. However, these features do not directly decide the vulnerabilities of the statements. Thus, removing them slightly declines the overall JIT-VL performance.

Moreover, the *operation* is an essential feature of the meaning of the analyzing statement. However, for each statement s, its *context$_c$* is obtained by slicing the CTG of the commit from s. Thus, the *context$_c$* contains not only related statements of s but also s itself. Therefore, even excluding the *operation* feature, this feature is still implicitly represented in *context$_c$*. Thus, the JIT-VL performance of the model is still maintained. However, if the *operation* is explicitly represented, it could enhance JULY's results 10% as seen in Table 7.

6.5. RQ5: Other factors

6.5.1. Impact of the imbalanced labels of the training data

To evaluate the impact of the data imbalance and the data balancing techniques, we employ two sampling techniques, under-sampling and over-sampling, to re-balance the class distribution in the training set while the testing set is left in the original imbalanced ratio. For under-sampling, we randomly delete examples from the majority class (non-vulnerable statements). For over-sampling, we randomly duplicate examples from the minority class (vulnerable statements). However, due to the limitation of computational resources, we cannot conduct the experiments with the over-sampling technique for the whole training set, i.e., the training set after over-sampling will have about 1.8M samples. To estimate how the

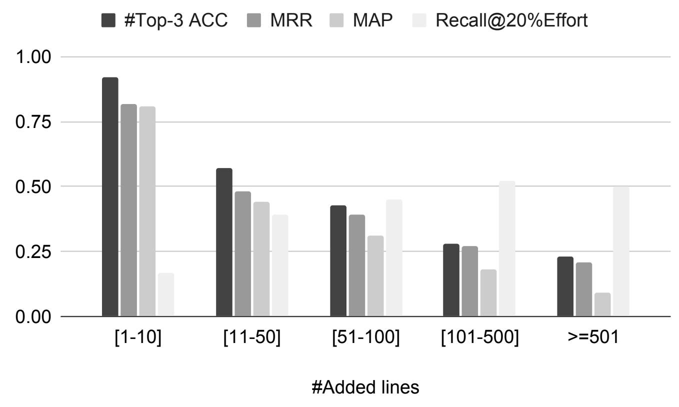


Figure 7: RQ5 – Impact of commit length on July's performance

over-sampling technique affects July's performance, we randomly over-sample the minority class three times and then under-sample the training set to obtain balanced samples for two labels. Specifically, the training set will contain about 100k samples for each class.

Table 8 shows that balancing the classes in the training set improves the overall performance of July. Specifically, Top-1 ACC of July is improved 90%. With the original imbalanced data, July is biased towards the majority class, and it only correctly ranks the vulnerable statements first in 10% of the commits. Meanwhile, with under-sampling, July could be trained to focus equally on both classes and thus better recognize the vulnerable statements. By training with balanced data, it can correctly localize vulnerable statements first in 19% of the commits.

Similar to under-sampling, in the over-sampling setting, July is also trained with balanced data, and its performance is also improved 1.7 times. Specifically, with over-sampling, July obtains 17% for Top-1 ACC. Meanwhile, with the original training dataset, this figure is only 10%. Interestingly, the performance of July in under-sampling and over-sampling techniques is stable. This shows that duplicating vulnerable statements does not prove new patterns to enhance the performance of July.

6.5.2. Impact of commit length

Figure 7 shows the impact of the number of added lines in a commit on July's performance. As seen, the smaller the number of added lines, the better July's performance. Specifically, for the commits containing 1–10 added lines, July's Top-3 ACC is about 92%. However, this result is declined two times if the number of added lines in the commits is 51–100 statements. This is reasonable because the larger a commit, the more difficult to understand all of its statements and distinguish them. July's accuracy could drop up to 4 times for extremely large commits, i.e., more than 501 statements.

6.6. RQ6: Time complexity

In this work, all experiments were conducted on an Ubuntu 18.04 server equipped with an NVIDIA Tesla P100 GPU. For the JIT-VD phase, July took about 5 hours to train an RGCN model. For the JIT-VL phase, July took about 2 hours to complete training the models. Moreover, JULY spends about 1.42 seconds to classify a commit as vulnerable or not. Also, identifying vulnerabilities within a commit took 0.12 seconds per commit.

6.7. Threats to validity

There are three main threats to the validity of our work:

Threats to internal validity: The primary threat related to the process of constructing CTG for the JIT-VD phase and extracting features for JIT-VL phase. JULY utilizes multiple program analysis techniques like structural analysis and program slicing to construct CTGs and derive corresponding features. To mitigate this threat, we employ the widely used code analyzer, Joern [29], and we carefully review our implementation.

Threats to construct validity: A threat might be associated with our evaluation procedure. To minimize this threat, we select commonly used metrics for evaluating approaches such as Top-K ACC, MRR, MAP, and Recall@20%Effort, which are consistent with established practices in related studies [21, 23, 24, 30].

Threats to external validity: A potential threat arises from the accuracy of vulnerability labeling within the dataset. It is possible that there are some mislabeled samples. To mitigate this, we select the large dataset, which is carefully built from investing the fixing-vulnerability commits in the public corpus [20]. Additionally, our experiments are conducted on only C/C++ programs. Thus, the results could not be claimed for other programming languages.

7. CONCLUSION

The surge in software vulnerabilities and the occurrence of numerous severe attacks in recent times have posed a challenge for Software Quality Assurance. In this paper, we introduce a novel framework, JULY, which serves a dual purpose: detecting vulnerable commits and fine-grained localizing the vulnerable statements. The core concept of JULY is that the vulnerability of a commit is primarily determined by the meaning of the code changes. To carry out JIT-VD, JULY represents each commit by a CTG and leverages an RGCN model to capture the meaning within these changes, thereby distinguishing between vulnerable and non-vulnerable commits. Once a commit is detected as suspicious, it is then passed to the JIT-VL model to precisely identify the root causes. In the JIT-VL process, JULY encodes each statement using four features: *operation*, $context_c$, $context_o$, and *topic*. Subsequently, JULY calculates a suspiciousness score for each modified statement and ranks them based on these scores. To evaluate JULY's effectiveness, several experiments were conducted using a dataset comprising 20,274 commits from 506 C/C++ projects. Notably, JULY demonstrates a significant improvement of 95% in Top-1 ACC and 63% *MRR*. Furthermore, when analyzing the same portion (i.e., 20%) of modified statements in each commit, JULY is able to identify twice as many vulnerable statements within a given commit compared to state-of-the-art approaches.

REFERENCES

[1] "CVE details." [Online]. Available: https://www.cvedetails.com

[2] "DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction, author=Hoang, Thong and Dam, Hoa Khanh and Kamei, Yasutaka and Lo, David and Ubayashi, Naoyasu," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 2019, pp. 34–45.

[3] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in Python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[4] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection," *Information and Software Technology*, vol. 136, p. 106576, 2021.

[5] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[6] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "VELVET: A noVel Ensemble Learning approach to automatically locate VulnErable sTatements," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering.* IEEE, 2022, pp. 959–970.

[7] S. Elder, N. Zahan, R. Shu, M. Metro, V. Kozarev, T. Menzies, and L. Williams, "Do I really need all this work to find vulnerabilities?" *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–78, 2022.

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

[9] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR).* Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 608–620.

[10] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[11] D. Hin, A. Kan, H. Chen, and M. A. Babar, "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, 2022, pp. 596–607.

[12] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[13] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, pp. 2072–2106, 2016.

[14] R. A. Khan, S. U. Khan, H. U. Khan, and M. Ilyas, "Systematic literature review on security risks and its practices in secure software development," *IEEE Access*, vol. 10, pp. 5456–5481, 2022.

[15] Y. Li, S. Wang, and T. N. Nguyen, "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*, 2021, pp. 292–303.

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

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

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

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013*, Y. Bengio and Y. LeCun, Eds., 2013.

[20] S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, and H. D. Vo, "Code-centric learning-based just-in-time vulnerability detection," *arXiv preprint arXiv:2304.08396*, 2023.

[21] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "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*, 2022, pp. 672–683.

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

[23] C. Pornprasit and C. K. Tantithamthavorn, "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, 2021, pp. 369–379.

[24] F. Qiu, Z. Gao, X. Xia, D. Lo, J. Grundy, and X. Wang, "Deep just-in-time defect localization," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 5068–5086, 2021.

[25] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you? explaining the predictions of any classifier," in *Proceedings of The 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 1135–1144.

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

[27] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.

[28] M. Wang, Z. Lin, Y. Zou, and B. Xie, "Cora: Decomposing and describing tangled code changes for reviewer," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1050–1061.

[29] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[30] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, and S. Li, "Just-in-time defect identification and localization: A two-phase framework," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 82–101, 2020.

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

[32] X. Yang, S. Wang, Y. Li, and S. Wang, "Does data sampling improve deep learning-based vulnerability detection? yeas! and nays!" in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2287–2298.

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

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

Received on October 07, 2023
Accepted on February 07, 2024