Enhancing Security Patch Identification by Capturing Structures in Commits

Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin

Abstract—With the rapid increasing number of open source software (OSS), the majority of the software vulnerabilities in the open source components are fixed silently, which leads to the deployed software that integrated them being unable to get a timely update. Hence, it is critical to design a security patch identification system to ensure the security of the utilized software. However, most of the existing works for security patch identification just consider the changed code and the commit message of a commit as a flat sequence of tokens with simple neural networks to learn its semantics, while the structure information is ignored. To address these limitations, in this paper, we propose our well-designed approach E-SPI, which extracts the structure information hidden in a commit for effective identification. Specifically, it consists of the code change encoder to extract the syntactic of the changed code with the BiLSTM to learn the code representation and the message encoder to construct the dependency graph for the commit message with the graph neural network (GNN) to learn the message representation. We further enhance the code change encoder by embedding contextual information related to the changed code. To demonstrate the effectiveness of our approach, we conduct the extensive experiments against six state-of-the-art approaches on the existing dataset and from the real deployment environment. The experimental results confirm that our approach can significantly outperform current state-of-the-art baselines.

Index Terms—Security Patch Identification, Graph Neural Networks, Abstract Syntax Tree

1 Introduction

Recently, a critical zero-day vulnerability named Log4Shell [1] has attracted widespread attention in the security community. It exists in the widely used Java logging library Log4j [2], which has been deployed by millions of Java applications. Specifically, due to this vulnerability can be exploited by attackers and leads to a remote code execution (RCE) when Log4j is used for logging, it was rated as "very critical" and received a CVSS [3] score of 10 (the highest level) . Although Log4Shell threats the security of many applications severely, however, due to the widespread attention, it has been resolved in a timely manner. On the contrary, most vulnerabilities are fixed silently, especially in the "big code" era, where open source software (OSS) has reached an unprecedented amount. According to the data from SourceClear [4], 53% of vulnerabilities in open source libraries are not publicly disclosed with CVEs. Hence, if the widely used open source components have unknown security risks, even if these risks are fixed in a timely manner by the developers of these components, the software that has already integrated them may not be updated in time, which causes substantial financial and social damages. Therefore, it is essential to identify these silent security patches.

Security patch identification is a crucial yet far from the settled problem, manual verification is a straightforward but unrealistic solution. According to the official report [5] released by the largest global open source software (OSS) platform GitHub, 100 million projects have hosted on their platform in 2018. A huge amount of security related patches are committed daily for fixing, which makes the manual verification time-consuming and labor-intensive. Hence, automated security patch identification is an inevitable choice.

Conventional software analysis techniques that have been widely used in software vulnerability detection, such as static analysis [6] and dynamic analysis [7] are not appropriate for security patch identification, because these techniques cannot be applied to the partial code (i.e., the changed code) in a commit. Furthermore, they also cannot support analysing the description of a commit (i.e., commit message), which is a form of natural text.

Inspired by the great success of machine learning and deep learning techniques in various fields [8], [9], [10], many learning-based approaches are proposed for automated security patch identification. For example, Zhou et al. [11] proposed to extract the features from the commit message and bug reports and further utilized a stacking of six machine learning classifiers such as random forest [12] and SVM [13] to identify security issues. Another advanced deep learning based technique SPI [14] encoded both commit message and code changes with the BiLSTM encoder followed by the CNN layer to learn better representations. However, most existing works for patch identification only consider the changed code and commit message as a flat sequence of tokens, while ignoring the structure information hidden in the text. The structure information of the program has been widely demonstrated to improve various code-related tasks, such as software vulnerability identification [15], source code summarization [16], and function name prediction [17], [18]. Inspired by these works, Commit2vec [19] further extracted AST paths related to the changed code and fed them to the fully connected layer to learn the representation of the changed code to iden-

- *S. Liu is the corresponding author.*
- *B. Wu, S. Liu, R. Feng, J. Siow and S. W. Lin are with Nanyang Technological University, Singapore. E-mail: {bozhi001, shangqin001, jingkai001}@e.ntu.edu.sg, {rtfeng,shang-wei.lin}@ntu.edu.sg*
- *X. Xie is with Singapore Management University, Singapore. E-mail: xiaofei.xfxie@gmail.com*

tify security-relevant commits. Commit2vec has achieved promising results, however, it only utilized the simple fully connected layer for learning, which has a limited learning capacity. Furthermore, the commit message is also ignored in utilization for the detection.

To address the aforementioned challenges, in this paper, we propose our well-designed tool for security patch identification named E-SPI. Specifically, to capture the structure information of the commit, we design two separate encoders, AST-based code change encoder, which extracts the contextual AST paths related to the changed code with the BiLSTM encoder to learn the semantics of the changed code; graph-based commit message encoder, which constructs the commit message by the dependency graph with gated graph neural network (GGNN) to capture the token relations in the commit message. We further ensemble both encoders for the security patch identification. In addition, to capture the contextual information that related to the changed code, we propose to extract the AST paths that are not limited within the changed code (i.e., Within-Changes), but also include the paths related to the changed code (i.e., Within-Context) to enhance the code change representation produced by the code change encoder. Extensive experiments on the existing dataset and the real deployment environment have demonstrated the effectiveness of our approach. Specifically, E-SPI outperforms current state-of-the-art approaches significantly with 4.01% higher accuracy and 4.22% F1 score on the existing dataset, increases 6.03% accuracy and 7.38% F1 in the real deployment environment. To sum up, our main contributions are as follows:

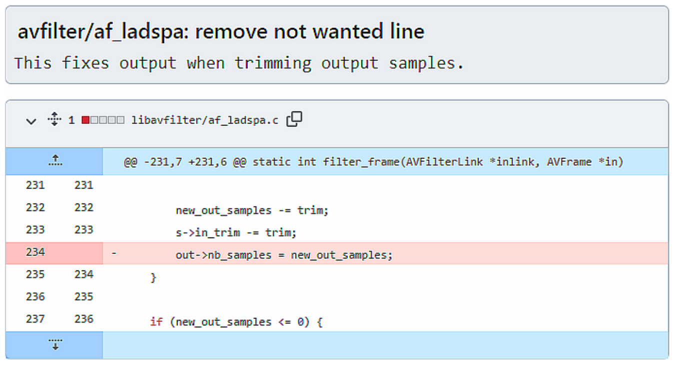
- We propose a novel approach (i.e., E-SPI) to capture the structure information of the commit to accurately learn the semantics for security patch identification.
- We design the contextual AST-based encoder to take the AST paths, which are not limited within the changed code, but further include the paths related to the changed code to capture the contextual information of the changed code.
- We conduct an extensive evaluation to demonstrate the effectiveness of E-SPI on the existing dataset and from the real deployment environment. The experimental results demonstrate that E-SPI significantly outperforms current state-of-the-art approaches. We have deployed our tool in industry for production.

2 Background

In this section, we briefly introduce the background of commit, abstract syntax tree (AST), and graph neural networks (GNNs) that we will use in this paper.

2.1 Commit

A commit, which usually consists of a commit message and a diff, records the modification between different software versions. Due to the various phases of the software development life cycle, a commit can serve for various purposes such as implementation logic, building configurations, and bug fixing. The commit message is used to illustrate the reasons for the modification in natural language, while the diff show the difference between different software versions (i.e., current vs previous), which may involve the modification of multiple files and functions.



avfilter/af_ladspa: remove not wanted line

```
This fixes output when trimming output samples.
```

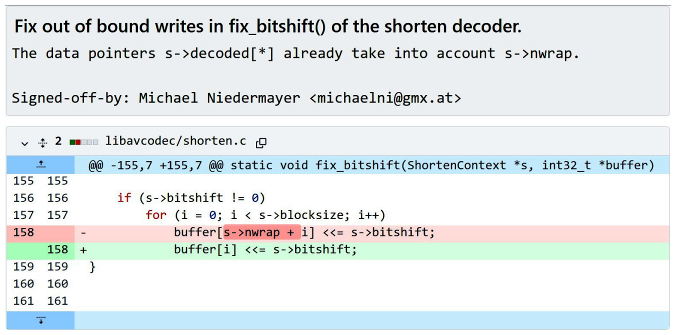
```
1 libavfilter/af_ladspa.c

@@ -231,7 +231,6 @@ static int filter_frame(AVFilterLink *inlink, AVFrame *in)

            new_out_samples -= trim;
            s->in_trim -= trim;
-           out->nb_samples = new_out_samples;
        }

        if (new_out_samples <= 0) {
```

Fig. 1: A non-security commit from FFmpeg with its commit id *ee9345e*.



Fix out of bound writes in fix_bitshift() of the shorten decoder.

```
The data pointers s->decoded[*] already take into account s->nwrap.

Signed-off-by: Michael Niedermayer <michaelni@gmx.at>
```

```
2 libavcodec/shorten.c

@@ -155,7 +155,7 @@ static void fix_bitshift(ShortenContext *s, int32_t *buffer)
155 155
156 156        if (s->bitshift != 0)
157 157            for (i = 0; i < s->blocksize; i++)
158     -              buffer[s->nwrap + i] <<= s->bitshift;
    158 +              buffer[i] <<= s->bitshift;
159 159    }
160 160
161 161
```

Fig. 2: A security commit from FFmpeg with its commit id *f42b319*.

We present a non-security and security sample commit from the open source project "FFmpeg" in Fig. 1 and Fig. 2 respectively to visually demonstrate their difference. To illustrate each component in a commit with more details, we choose the security commit in Fig. 2 as an example. The content in the upper rectangle is the commit message, which describes the purpose of the current modification in natural language. For this example, the modification serves to fix the bug that an array may be out of the boundary. The lower rectangle is the diffs. We can see that it consists of file name (i.e., "libavcodec/shorten.c") and the modified chunk from line 155 to 161 in file "shorten.c". The first line starting with "@@" in a chunk is to show some specific information, such as the start modified line number and the method name in the original file. The code change consists of the content marked with "+" in the updated file and its previous version marked with "-" in the original file to record the changed code. For example, there is a code change on line 158 in "libavcodec/shorten.c", from "buffer[s−> nwrap + i] <<= s −> bitshift;" to "buffer[i] <<= s −> bitshift;"in Fig. 2. The remaining contents in a chunk are the context about the code changes to reveal the contextual information for the changed code, such as the content from line 156 to line 157. Here, we use a simple commit with only one chunk as a sample for illustration. For other cases, a commit may consist of multiple chunks in one file or multiple changed files.

2.2 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code, while removing unnecessary information such as punctuation and delimiters (e.g., braces, semicolons, parentheses). Hence, AST can

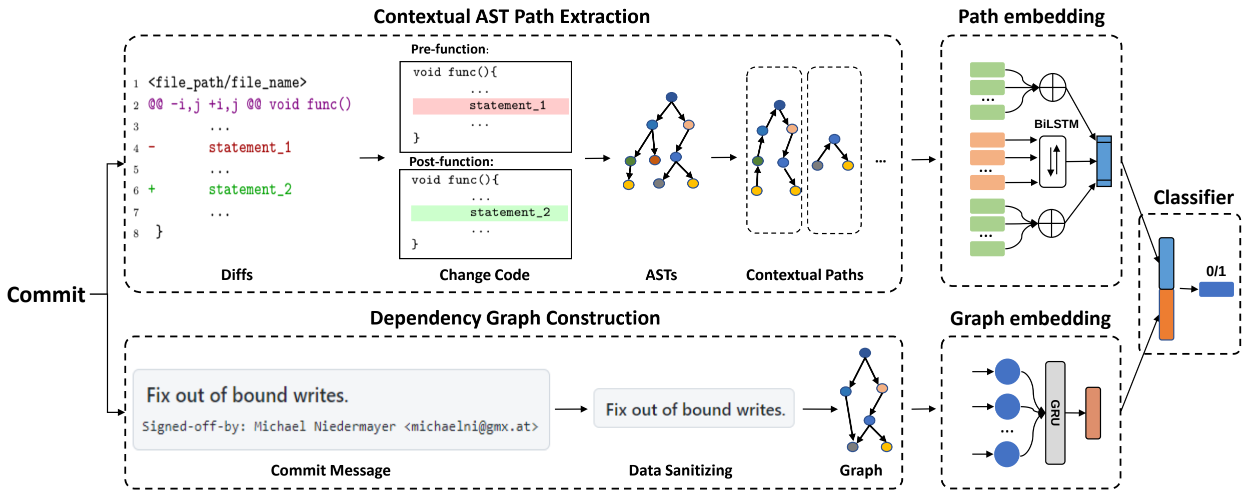


Fig. 3: The overview of our framework for security patch identification.

be regarded as a high-level abstraction of the source code, which allows program analysis to ignore the grammatical details and focus on the program semantics. A simple AST example is shown in Fig. 5, we can see that it has two types of nodes: terminal nodes and non-terminal nodes, where terminal nodes usually refer to user-defined tokens and represent the names of identifiers from the code such as "buffer", "i"; non-terminal nodes represent a limited set of syntactic structures in a programming language such as "if_stmt", "func_declarator". Since AST is an abstraction of the source code without redundant information, many works [17], [20], [21], [22], [18] utilize it as the program representation for various code-learning tasks.

2.3 Graph Neural Networks

Graph Neural Networks (GNNs) [23], [24], [25], [26] can model graph-structured data, which contain nodes and the relations among them, making them popular in different domains such as the social network [27], molecules [24] and the program [28]. GNNs utilize the message passing mechanism to communicate neighborhood information among different type of edges. Based on the different design of message passing, there are a variety of GNN variants such as gated graph recurrent network (GGNN) [23], graph convolutional network (GCN) [25] and graph attention networks (GAT) [29]. Since a program can be parsed into a structural graph with comprehensive semantics [30], [28], GNNs has achieved promising results on many code-learning tasks such as code summarization [20], [16], code search [31] and vulnerability detection [15].

3 Approach

In this section, we first formulate the security patch identification problem and then introduce an overview of our approach. We detail each component of our well-designed network for further illustration.

3.1 Problem Formulation

In this paper, similar to Zhou et al. [14], E-SPI aims at identifying a commit is a security patch or not. Formally, given a dataset $D = \{(c, y) | c \in \mathcal{C}, y \in \mathcal{Y}\}$, where $\mathcal{C}$ is a set of commits, $\mathcal{Y} \in \{0, 1\}$ is the set of labels with 1 to indicate that the commit is the security patch and 0 for the non-security patch. E-SPI formulates it as a binary classification problem and aims to learn a mapping function $f : \mathcal{C} \to \mathcal{Y}$ for automated identification. The prediction function f can be learned by minimizing the loss function as follows:

$$\min \sum_{i}^{n} \mathcal{L}(f(y_i | c_i)) \tag{1}$$

where $\mathcal{L}(\cdot)$ is the loss function and we choose the cross entropy to optimize the learned weights, n is the total number of commits in the training set.

3.2 Overview

The overview of E-SPI is shown in Fig. 3, we can see that it jointly utilizes diffs and commit message for the security patch identification. Specifically, to capture the context of the code changes, we first retrieve the corresponding modified functions about the code changes, which are defined as pre-functions and post-functions accordingly. Then, we extract the corresponding contextual AST paths related to the changed code on ASTs, which are parsed from these retrieved functions, to capture the structure information. We feed these extracted AST paths to a code change encoder to learn the representation for the changed code. For the commit message, E-SPI constructs the dependency graph to capture the token relations in the sequence and uses a commit message encoder to capture the structure information behind the text. Then, E-SPI concatenates the vector representations produced by the code change encoder and the commit message encoder with a fully connected layer to predict whether the commit is a security patch or not.

3.3 AST-based Code Change Encoder

Most of the existing works [14], [32] for security patch identification ignore the structure information hidden in the changed code, which limited to achieve the promising results. In addition, they mostly relied on the changed code

```c
static void fix_bitshift(ShortenContext *s, int32_t *buffer)
{
    int i;

    if (s->bitshift != 0)
        for (i = 0; i < s->blocksize; i++)
            buffer[s->nwrap + i] <<= s->bitshift;
}
```

(a) Pre-function f_s.

```c
static void fix_bitshift(ShortenContext *s, int32_t *buffer)
{
    int i;

    if (s->bitshift != 0)
        for (i = 0; i < s->blocksize; i++)
            buffer[i] <<= s->bitshift;
}
```

(b) Post-function f_a.

Fig. 4: The complete function f_s and f_a extracted from the commit in Fig. 2.

(e.g., the statement of the line marked with "+" or "-" in Fig. 2) for embedding, while the contextual information related to the changed code is missed. To overcome these limitations, we propose a well-designed AST-based code change encoder which consists of two sequential parts (i.e., contextual AST path extraction and path embedding).

3.3.1 *Contextual AST Path Extraction*

To capture the contextual information for the changed code, we first retrieve the complete functions changed by the commit. Specifically, we extract the corresponding subtractive statements marked with "-" and define them as d_s. Similarly, we also extract the additive statements marked with "+' and define them as d_a. Based on d_s and d_a, by line number and file name, we can retrieve their original functions defined as the pre-function f_s and the post-function f_a. For example, considering the commit in Fig. 2, d_s and d_a are "buffer[s−> nwrap + i] <<= s −> bitshift;" and "buffer[i] <<= s −> bitshift;", respectively, with the given line number (e.g., 158) and file name (e.g., "bavcodec/shorten.c"), we can successfully retrieve their original functions f_s and f_a, which are shown in Fig. 4.

Inspired by Code2Vec [18] and Code2Seq [17], which parsed the function into AST and constructed the paths between the start node and end node (both of them are the terminal nodes on AST) to capture the structure information behind the function text, we also convert the original functions (i.e., f_s and f_a) into ASTs by adopting *tree-sitter* to construct the AST paths. However, unlike their work, which focused on the function-level data, our approach is customized to construct the paths for commits. Specifically, the start node is located at the changed code, while the end node comes from different places based on specific considerations. Furthermore, the end node can be categorized into two groups:

- Within-Changes: We include the end node in the changed code to capture the structure information of the changed code. For example, as shown in Fig. 4a, the subtractive statement contains two variables "s" and "bitshift", hence, there is an AST path[1] starts from the "s" and ends at the "bitshift", which is marked with green arrows in Fig. 5.
- Within-Context: To capture the contextual information for the changed code, we randomly select the end nodes, which are from the context for the enhancement. Here, we define the contextual information as AST paths where the start node belongs to the changed code and the end node is from other code in a function. For example, there is a variable "blocksize" and it is not in d_s, however, it is included in the context of d_s, which is shown in Fig. 4a, we also construct a path between the "s" and "blocksize", which is marked with blue arrows in Fig. 5.

1. The shortest path between the start node and the end node.

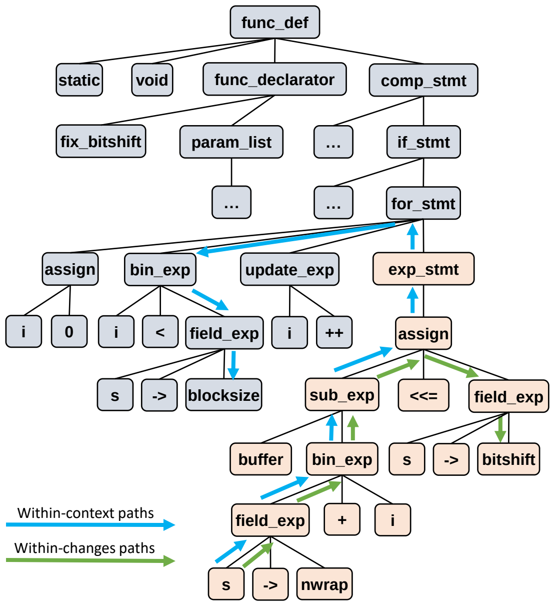


Fig. 5: The extracted AST paths from within-changes and within-context, where the function is plot in Fig. 4a.

For any commit c, we first extract AST paths of subtractive statements and additive statements, respectively. Then we randomly sample a set of AST paths $\{x_1, \cdots, x_k\}$ from within-changes paths and within-context paths at a ratio of 1:1, where k is their total number. We will discuss the value of k and the ratio in the discussion part of Section 6.

3.3.2 *Path Embedding*

Given a set of AST paths $\{x_1, \cdots, x_k\}$, each path can be defined as $x_i = v_1^i v_2^i \cdots v_l^i$, where l indicates the total number of nodes, v_1^i and v_l^i are the start node and end node of the i-th path in the AST path set. The start node and the end node are the terminal nodes of AST, which consists of node type and node value, while the non-terminal nodes in the AST path just have the node type. We separately encode the AST path and the terminal nodes as shown in Fig. 6.

- Path Representation: There are a total number of 220 different symbols produced by *tree-sitter* to represent the node type in AST. However, some of the types may be composed of more than one token, such as "function_definition". To handle this type of cases, we simplify them to abbreviations. For example, the node type "function_declaration" is simplified to "FuncDef". After that,

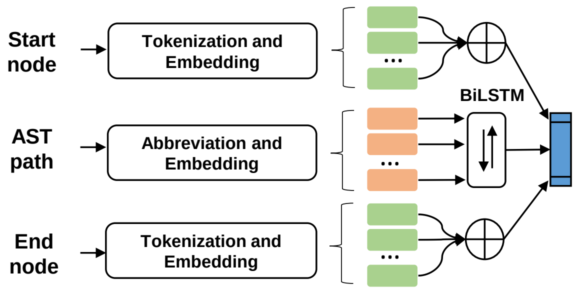

Start node
Tokenization and Embedding
AST path
Abbreviation and Embedding
End node
Tokenization and Embedding
BiLSTM

Fig. 6: The architecture of the path embedding.

we use a learnable embedding matrix $\boldsymbol{E}^{\text{type}} \in \mathbb{R}^{v_1 \times d}$ to encode each node type, where v_1 is the length of the vocabulary set for the node type, d is the dimensional length, and feed them into the bidirectional LSTM (BiLSTM) to obtain the final hidden states, which can be formulated as follows:

$$\boldsymbol{h}_{v_1^i}, \cdots, \boldsymbol{h}_{v_l^i} = \text{BiLSTM}(\boldsymbol{E}^{\text{type}}_{v_1^i}, \cdots, \boldsymbol{E}^{\text{type}}_{v_l^i}) \tag{2}$$

$$\boldsymbol{r}_{path} = [\boldsymbol{h}^{\rightarrow}_{v_l^i}; \boldsymbol{h}^{\leftarrow}_{v_1^i}] \tag{3}$$

where l is the total number of nodes in an AST path.

- Token Representation: The node v_1^i and v_l^i in x_i are terminal nodes and their values are tokens from the diffs. We split the values into a set of subtokens $S_{subtoken}$ to reduce the vocabulary set. We further represent each subtoken with another learnable embedding matrix $\boldsymbol{E}^{\text{subtokens}} \in \mathbb{R}^{v_2 \times d}$, where v_2 is the length of the vocabulary set for the subtokens, and then sum them up to represent the terminal nodes. For example, for the node v_1^i, the calculation can be represented as follows:

$$\boldsymbol{r}_{v_1^i} = \sum_{s \in S_{\text{subtoken}}} \boldsymbol{E}^{\text{subtokens}}_s \tag{4}$$

Finally, we concatenate the vector representation of start node (i.e., $\boldsymbol{r}_{v_1^i}$) and end node (i.e., $\boldsymbol{r}_{v_l^i}$) with the AST path (i.e., $\boldsymbol{r}_{path}$), and apply a fully-connected layer followed by a layer normalization to obtain the embedding of the entire AST path.

$$\boldsymbol{v}_i = \text{Norm}(\text{FC}([\boldsymbol{r}_{v_1^i}; \boldsymbol{r}_{path}; \boldsymbol{r}_{v_l^i}])) \tag{5}$$

Since the code changes are represented by a set of k AST paths, we apply the max-pooling operation over the embedding vector of the AST paths i.e., $\boldsymbol{v}_c = \text{maxpool}(\{\boldsymbol{v}_i\}_{i=1}^k)$ to get the final representation.

3.4 Graph-based Commit Message Encoder

Apart from the changed code, a commit also contains a commit message, which illustrates the purpose of the current changes. Compared with existing approaches [14], [32], which feed the message into a BiLSTM module to capture the sequential dependency in the sequence, E-SPI innovates it by constructing the dependency graph with the graph neural network to capture the relations between tokens in the sequence.

3.4.1 Dependency Graph Construction

The dependency parse tree [33] describes the dependency relations among words in a sentence by directed linked edges, which has been widely used in word relation extraction [34], [35] in NLP. To better learn the semantics in the commit message, we construct the dependency graph with two steps: data sanitization and graph construction. For data sanitization, since software developers may record some information not related to the code changes, such as website links, signatures, or emails. To sanitize the commit message, we remove these irrelevant information in commit message by the regular expression. Then, we construct the dependency graph according to the cleaned commit message.

Specifically, given the cleaned commit message m consists of multiple sentences, for each sentence in m, we utilize *Spacy* to construct its dependency tree. We further connect the neighboring dependency parse trees by connecting those nodes that are at the end and the beginning of two adjacent sentences with the edge "neigh" and construct the dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes in the graph and $\mathcal{E}$ denotes the relations (i.e., edges) within the nodes. Furthermore, each node in the graph is also the original token in m and there are 49 different edge types to describe these token relations. We take the commit message in Fig. 2 as an example and the constructed dependency graph is shown in Fig. 7. We can see that a token may have different semantic relations linked to other tokens. For example, there is an edge "prep" points from "fix" to "out" and "of" to describe the prepositional modifier of the verb "fix". Furthermore, to connect two different sentences, we further add an edge "neigh". For example, as shown in Fig. 2, the edge "neigh" connects the nodes that are the last token "." in the first sentence and the first token "the" in the second sentence.

3.4.2 Graph Embedding

Given the dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to describe the relations in the tokens, we need to learn its representation. A variety of graph neural networks (GNNs) [23], [29], [25] can be chose, inspired by the recently advanced Gated Graph Neural Network on modelling the program [28], [36], [16], [31], [37], we also select GGNN to learn the vector representation, as shown in Fig. 8. Specifically, each node $v \in \mathcal{V}$ is initialized by a learnable embedding matrix $\boldsymbol{E} \in \mathbb{R}^{v_3 \times d}$, where v_3 is the length of the vocabulary set for the commit message and d is the dimensional length, and gets its initial vector $\boldsymbol{h}_v^0 \in \mathbb{R}^d$. We apply a fix number of hops (i.e., T) to propagate the node information along the edges. At each computation hop t, where $1 \leq t \leq T$, we select the summation function as the aggregation function to aggregate the neighboring node features from the previous hop, which can be expressed as follows:

$$\boldsymbol{h}^t_{N(v)} = \text{SUM}(\{\boldsymbol{h}_u^{t-1} | \forall u \in N_{(v)}\}) \tag{6}$$

where $N_{(v)}$ is a set of neighborhood nodes that are connected with v. Then, we use a Gated Recurrent Unit (GRU) [38] to update the node representation as follows:

$$\boldsymbol{h}_v^t = \text{GRU}(\boldsymbol{h}_v^{t-1}, \boldsymbol{h}^t_{N_{(v)}}) \tag{7}$$

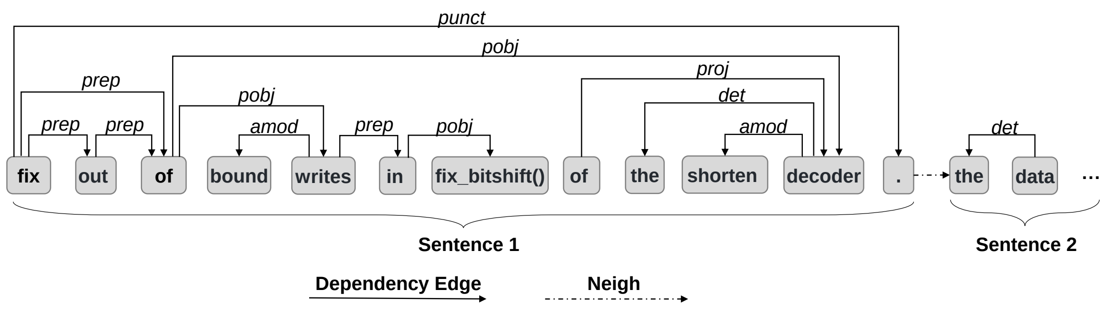


Fig. 7: The constructed dependency graph for the commit message in Fig. 2.

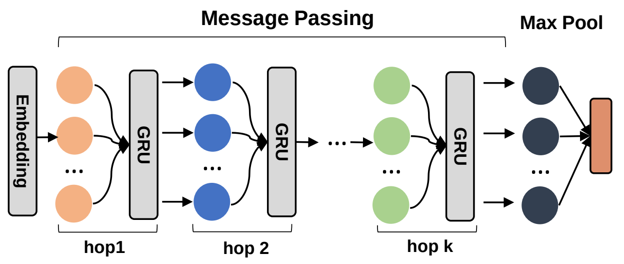


Fig. 8: The architecture of the graph embedding.

After a fixed number of hops (i.e., T), we obtain the final node representation $\boldsymbol{h}_v^T$ and apply the max-pooling over all nodes $\{\boldsymbol{h}_v^T | \forall v \in V\}$ to obtain the graph representation $\boldsymbol{v}_m$, defined as follows:

$$\boldsymbol{v}_m = \mathrm{maxpool}(\mathrm{FC}(\{\boldsymbol{h}_v^T | \forall v \in V\})) \tag{8}$$

where FC is a fully connected layer. The graph representation $\boldsymbol{v}_m$ can be considered as the vector representation for the cleaned commit message m.

3.5 Ensembling

By the code change encoder and commit message encoder, we can get the vector representation for the changed code (i.e., $\boldsymbol{v}_c$) and commit message (i.e., $\boldsymbol{v}_m$) respectively. We further ensemble both to represent the commit c. Specifically, we concatenate the graph embedding $\boldsymbol{v}_m$ with the code change embedding $\boldsymbol{v}_c$, and apply a fully connected layer to the classifier:

$$\mathrm{prob} = \mathrm{Sigmoid}(\mathrm{FC}([\boldsymbol{v}_m; \boldsymbol{v}_c])) \tag{9}$$

$$\mathrm{prediction} = \begin{cases} 1 & \text{if prob} \geq 0.5 \\ 0 & \text{if prob} < 0.5 \end{cases} \tag{10}$$

where prob denotes the probability obtained by the activation function Sigmoid, prediction is the predicted result where 0 for prob less than 0.5 and 1 for the others.

4 Evaluation Setup

In this section, we first introduce the dataset used, followed by the selected baselines for the comparison, then present the evaluation metrics with the model settings of E-SPI.

TABLE 1: The statistics of dataset

Project	Security	Non-security	Total
Linux	7,358	3,144	10,502
FFmpeg	5,393	6,210	11,603
QEMU	4,256	5,516	9,772
Wireshark	2,889	3,827	6,716
Total	19,896	18,697	38,593

4.1 Dataset

Currently, to the best of our knowledge, in addition to SPI [14] making their data partially public (i.e., FFmpeg and QEMU) are available, the other works for security patch identification have not released the data so far. Hence, to facilitate the evaluation, we borrow the SPI dataset, which is collected from 4 high-profile open source projects (i.e., Linux, FFmpeg, Wireshark, QEMU) on Github with the help of manual analysis and labeling by security analysts. After filtering out those samples that cannot be parsed into ASTs or graphs, we finally get the dataset for evaluation. As shown in Table 1, the dataset consists of 38,593 labeled commits, where 19,896 of them are security related patches and 18,697 are non-security. Furthermore, the commits of Linux are collected from 2016 to 2017, while the other three projects are up to January 2018. Different from SPI, we split the data of each project with the ratio of 80:10:10 and then fuse the divided data of each project as train/validation/test sets for evaluation.

4.2 Baselines

To evaluate our approach, we compare E-SPI with six state-of-the-art security patch identification approaches. The details of these six approaches are shown as follows:

Stacking [11]. Zhou et al. proposed to identify security issues from commit messages and bug reports by learning a stack of six basic classifiers such as SVM, random forest for automated identification. In E-SPI, since the bug reports are not available, we ignore them and just use the commit messages for the experiments.

SPI [14]. SPI considered both commit messages and code changes as the input. It proposed two separate encoders to learn the representation of commit message and code change respectively and each of them consists of a BiLSTM layer and a CNN layer. Then, two encoders are combined together to get the final probability of predicting whether a commit is vulnerable or not.

PatchRNN [32]. Similar to SPI, PatchRNN also considered two separate BiLSTM encoders to detect security patches. In addition to commit message, it takes the entire function code before and after the patch into consideration, rather than the code changes used in SPI.

Commit2vec [19]. It ignored the commit message and just used the code changes to identify the security-relevant commit, Furthermore, it extracted AST paths, which are related to the code changes as the input to represent the code changes, and fed the paths into the fully connected layers to learn the vector representation for identification.

Transformer [39]. Transformer has achieved great success in different fields. We also include it as a baseline. Specifically, we utilize two separate transformer encoders to encode the commit message and code changes respectively. Each encoder has 6 identical layers, each of them has 8 heads to learn different subspace features.

VulFixMiner [40]. VulFixMiner applied CodeBert to extract semantic meaning from code changes to identify silent vulnerability fixes. Similar to commit2vec, VulFixMiner did not consider commit message.

Except for SPI, which we reproduce the experiments with the official code, none of the other baselines are open source, and we re-implement these baselines following to the original papers strictly and carefully.

4.3 Evaluation Metrics

Following SPI [14], we use **precision** (Pre), **recall** (Rec) and **F1** as our evaluation metrics, and further add **accuracy** (Acc), since it is a common metric to evaluate classification systems. Before introducing the used metrics, we first illustrate four important terms that will be used in computing the metric values:

- true_positives: the cases in which predicted as security patches actually are security patches.
- true_negatives: the cases in which predicted as non-security patches actually are non-security patches.
- false_positives: the cases in which predicted as security patches actually are non-security patches.
- false_negatives: the cases in which predicted as non-security patches actually are security patches.

Precision represents the proportion of true positives to the predicted total positives, which infers how precise the model is in identifying security patches. The formula of precision is shown below.

$$\text{precision} = \frac{\text{true\_positives}}{\text{true\_positives} + \text{false\_positives}} \tag{11}$$

Recall is the ratio of true positives to all positives in ground truth, which can be seen as a measure of how robust the model is in identifying security patches. It can be calculated as follows.

$$\text{recall} = \frac{\text{true\_positives}}{\text{true\_positives} + \text{false\_negatives}} \tag{12}$$

F1 is the harmonic mean between precision and recall. A high F1 implies low false positive as well as low false negative. In other words, a high F1 means that the model is highly precise and robust. F1 can be computed as follows.

$$\text{F1} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{13}$$

Accuracy is the ratio of the number of correct predictions to the total number of input samples. Therefore, a high accuracy implies that the model performs well in identifying both security patches and non-security patches. Accuracy can be formulated as follows.

$$\text{accuracy} = \frac{\text{true\_positives} + \text{true\_negatives}}{\text{total\_sample}} \tag{14}$$

where total_sample is the sum of true positives, true negatives, false positives and false negatives.

4.4 Model Settings

For the code change encoder, we set the embedding size for the node type and node value to 128. The dimension of hidden states in the BiLSTM is set to 128. The maximum number of AST paths k is set to 500 to represent the code changes. For the commit message encoder, the embedding size of tokens is also set to 128, following the code change encoder. Furthermore, the dimension size of the hidden states in GRU is 128. We use the Adam optimizer with a learning rate of 0.001 to train the model. The early stop is set to 10 which means that the training process is stopped when no improvements on Acc for 10 epochs. All experiments are conducted on Intel(R) Xeon(R) Gold server with three Nvidia Graphics Tesla V100. To avoid the effect of fluctuations on the random seed, we repeat the experiments three times with different random seeds and report the average values for all experiments.

5 Evaluation Results

To evaluate the effectiveness of E-SPI, we first compare the performance of E-SPI against the state-of-the-art approaches and then conduct a deep analysis on the effect of contextual AST paths and graph-based commit message encoder. To confirm the validity of E-SPI, we further investigate its performance in a real deployment environment and provide a qualitative analysis of the prediction results.. Our experiments mainly focus on the following five research questions:

- RQ1: Can E-SPI outperform the baselines in the evaluation metrics?
- RQ2: Does the contextual AST paths improve the performance in code change encoder?
- RQ3: Is the graph-based commit message encoder powerful in learning the commit message?
- RQ4: What is the performance of E-SPI in a real deployment environment?
- RQ5: The qualitative analysis of the results provided by E-SPI.

5.1 RQ1: Comparison with the State-of-the-art Approaches

We compare E-SPI with six state-of-the-art approaches and the evaluation results are shown in Table 2.

An interesting finding is that the machine learning-based approach Stacking significantly outperforms the deep learning-based approach Commit2vec and VulFixMiner, which demonstrates that commit messages play a more

TABLE 2: Evaluation results in percentage compared with the baselines.

Approach	Acc	Pre	Rec	F1
Stacking	69.35	71.50	67.44	69.41
Commit2vec	62.29	62.75	66.26	64.46
PatchRNN	81.68	83.05	81.01	82.01
SPI	84.95	85.82	84.82	85.32
Transformer	74.27	78.37	69.20	73.50
VulFixMiner	63.61	63.25	65.23	64.22
E-SPI	**88.96**	**87.88**	**91.26**	**89.54**

critical role than code changes for the security patch identification and performance by just using a stack of multiple machine learning classifiers on the messages can outperform the performance of deep neural networks on the code changes. After analysis, we find that when developers fix a bug and submit the corresponding patch, they would describe their purpose in detail in commit message. For security patches, their commit message often contains some security-related words, such as ”fix buffer overflow” and ”fix memory leak”, which contain explicit security-related information and are easily recognized by the model. In contrast to commit messages, security information in code changes is implicit and cannot be easily extracted from the source code. Therefore, having commit messages as input can bring more benefits than code changes.

Furthermore, we find that the performance of Transformer is lower than that of BiLSTM-based models (i.e., PatchRNN and SPI), which is in conflict with the current consensus that transformer has more expression capacity than BiLSTM. We conjecture that it is caused by the limited amount of data used for training. Specifically, the total number of samples in the dataset is only 40,523, which makes transformer overfit to the training data. The smaller size of the dataset impacts the transformer to obtain the promising results compared with the BiLSTM-based models.

The last row in Table 2 shows the results of E-SPI. We can observe that E-SPI outperforms the baselines significantly, achieving 88.61% in accuracy and 89.25% in F1, exceeding the best baseline PatchRNN by 5.99% in accuracy and by 5.81% in F1. The experimental results demonstrate the effectiveness of our designed AST-based code change encoder and graph-based commit message encoder for security patch identification.

Answer to RQ1: E-SPI outperforms six state-of-the-art baselines significantly in accuracy and F1, we attribute the improvements to the well-designed code change encoder and the commit message encoder.

5.2 RQ2: Comparison with Different Code Change Encoders

We further conduct experiments to compare the performance of different code change encoders to confirm the effectiveness of our designed AST-based code change encoder. Specifically, we compare with PatchRNN, SPI, Transformer, VulFixMiner and Commit2vec, where PatchRNN took the entire function code as input with BiLSTM for learning, SPI fed the code changes with a BiLSTM followed by the CNN layer to learn the representation, transformer fed the changed code into the transformer encoder for learning, VulFixMiner fed the code changes into a CodeBert to get the representation, and Commit2vec extracted the AST paths related to the code changes and fed these paths into fully-connected layers for learning. For these approaches, if they involve the commit message encoder, we turn it off and just use the code change encoder to compare the results. In addition, we also ablate the effect of the AST paths that only extracted from the code changes (i.e., E-SPI w/o Context) and the paths where the end node is from the context (i.e., E-SPI w/o Changes). The experimental results are presented in Table 3.

We can observe that our AST-based code changes encoder outperforms the baseline code changes encoders by a significant margin in terms of Acc and F1. Specifically, PatchRNN performs the worst among these approaches, we infer that it is caused by PatchRNN takes the entire function rather than the changed code as the input, which introduces too many noises for patch identification. Considering purely using the changed code for learning, transformer has a better performance than SPI. However, when the commit message encoder is turned on, the performance is lower than SPI in Table 2. We believe it is reasonable because the difficulty of learning the natural language commit message is much lower than the code, which makes the equipped powerful transformer overfit to the commit message encoder. Thus, when combining both encoders, it performs worse than BiLSTM. However, when purely using the transformer for code learning, it has a higher performance than BiLSTM due to the more powerful expression ability of the transformer. VulFixMiner performs close to Transformer, as it also leverages a powerful model for patch identification. However, the performance of the transformer and VulFixMiner is still lower than E-SPI, we attribute to the structure information used in the changed code of E-SPI. Furthermore, although Commit2vec has higher accuracy and F1 than SPI, due to that it also encodes AST as input, its performance is lower than E-SPI. We believe that it is caused by the simple fully connected layer used in Commit2vec that has limited learning capacity compared to BiLSTM used in E-SPI.

Lastly, we ablate the effect of the contextual information used in AST paths and the experimental results are shown at the last row of Table 3. We can see that the paths extracted from the changed code and the context are both beneficial in improving the performance, and when incorporating both, E-SPI could achieve the best performance.

Answer to RQ2: AST-based code change encoder achieves higher performance as compared to the baseline encoders, we attribute to the structure information used in the changed code. Furthermore, we also confirm that the contextual AST paths are also beneficial in improving the performance.

5.3 RQ3: Comparison with Different Commit Message Encoders

Similar to Section 5.2, we also make a comparison with four state-of-the-art commit message encoders, including Stacking, PatchRNN, SPI, and Transforme, where Stacking used a stack of 6 basic machine learning classifiers on the

TABLE 3: Evaluation results on different code change encoders.

Approach	Acc	Pre	Rec	F1
PatchRNN	59.07	61.43	55.38	58.25
SPI	61.37	61.86	65.34	63.56
Transformer	62.71	60.84	77.65	68.23
Commit2vec	62.29	62.75	66.26	64.46
VulFixMiner	63.61	63.25	65.23	64.22
E-SPI w/o Changes	64.94	**65.14**	69.61	67.30
E-SPI w/o Context	63.49	64.02	66.81	65.39
E-SPI	**65.56**	63.59	**78.27**	**70.17**

TABLE 4: Evaluation results on different commit message encoders.

Approach	Acc	Pre	Rec	F1
Stacking	69.35	71.50	67.44	69.41
PatchRNN	81.84	84.41	79.45	81.85
SPI	83.92	84.98	83.58	84.27
Transformer	79.43	82.86	75.78	79.16
E-SPI	**87.66**	**85.53**	**91.66**	**88.49**

extracted features of commit messages, PatchRNN encoded the commit message with a BiLSTM to learn the representation, SPI further added a CNN layer followed by BiLSTM for classification, and Transformer fed the message to the transformer encoder for learning. The experimental results are shown in Table 4.

We can see that Stacking has the worst performance. We attribute to the limitation by the used techniques (i.e., the adopted machine learning techniques) are less advanced than deep neural networks for the same input. Furthermore, transformer has a lower performance than PatchRNN and SPI, which confirms that transformer is easier to overfit to the commit message and thus has a lower performance. Due to the lower performance of the transformer compared to PatchRNN and SPI in the commit message, it further affects its performance when combined with the code change encoder (see Table 2). Finally, we can find that E-SPI achieves the highest accuracy, precision, recall and F1 at 87.26%, 85.04%, 91.66% and 88.23%, which outperforms other approaches significantly. It indicates that by constructing the dependency graph to capture the structure information behind the commit message and further utilizing GNNs to learn the token relations, our designed commit message encoder could learn the semantics well, and thus produces promising results.

Answer to RQ3: Graph-based commit message encoder is powerful to produce the promising results, we attribute to the constructed dependency graph with GNNs to learn the structure information behind the text to capture the commit message semantics.

5.4 RQ4: Evaluation in a Real Deployment Environment

We discuss the efficiency and practicality of E-SPI in a real production setting. We deploy E-SPI with the baselines on an industrial platform from our industry collaborator and investigate the real performance in terms of detection accuracy and efficiency. Specifically, we deployed the pipeline of each approach to the real-time platform, and each pipeline

TABLE 5: Evaluation results on unseen commits from the deployment.

Approach	Acc	Pre	Rec	F1
Stacking	58.45	65.81	51.35	57.69
Commit2vec	55.65	59.42	54.53	56.87
PatchRNN	81.57	90.53	74.36	79.05
SPI	81.87	**91.51**	73.99	81.82
Transformer	67.27	63.61	**95.01**	76.20
VulFixMiner	56.13	58.65	56.03	57.31
E-SPI	**86.81**	84.49	91.52	**87.86**

consists of the scripts of data pre-processing for commits, the pre-trained model with scripts for prediction.

We first evaluate the accuracy among different approaches on the unseen commits labeled by professional security experts from industry. As shown in Section 4.1, the dataset used consists just of commits before January 2018. Hence, we further collect the commits from February 2018 to March 2021 from the four open source projects (that is, Linux, FFmpeg, QEMU, and Wireshark) with the help of four security experts. Following SPI [14], the keyword filtering process and cross-validation by experts are carried out to filter irrelevant commits. After the strict validation, there are 6,297 samples in total, of which 3,517 are the security related patches and 2,780 are non-security for the four projects. We test the performance of different approaches on these data and show the experimental results in Table 5. We can observe that all approaches suffer from a decline over the evaluation metrics, compared with the results in Section 5.1. It is reasonable, since the diversity of the newly collected samples is increased in the real deployment environment. In contrast, for a specific dataset, where the data distribution over this dataset is fixed, the model tends to perform better compared to deploying it to the unseen data. However, although performance is decreased on these approaches, E-SPI could still achieve better performance in terms of accuracy and F1 compared to other baselines, which confirms the effectiveness of our approach.

To evaluate detection efficiency, we average the prediction time, which includes the data pre-processing time and inference time, for the collected commits, and the time spent for predicting one commit by each approach is presented in Table 6, where the "Extraction" column defines the time for the commit-related function extraction, "Processing" column defines the processing time to convert the data into an appropriate format for the model, "Inference" column defines the time for model inference and "Total" column is the sum time for the previous three columns. Because the approach Stacking, SPI, Transformer and VulFixMiner only utilize changed code as the input, the time for the function extraction is zero. From Table 6, we can see that E-SPI requires longer time to make a prediction, which indicates the higher complexity of our approach. Furthermore, we can find that the pre-processing occupies most of the time in our approach, since it is relatively complicated and involves many operations such as AST path extraction and message dependency graph construction. However, generally, we can still conclude that the time for a prediction by our approach is acceptable since it only takes around 0.2 second to produce a result.

TABLE 6: Time spent in millisecond (ms) by each approach.

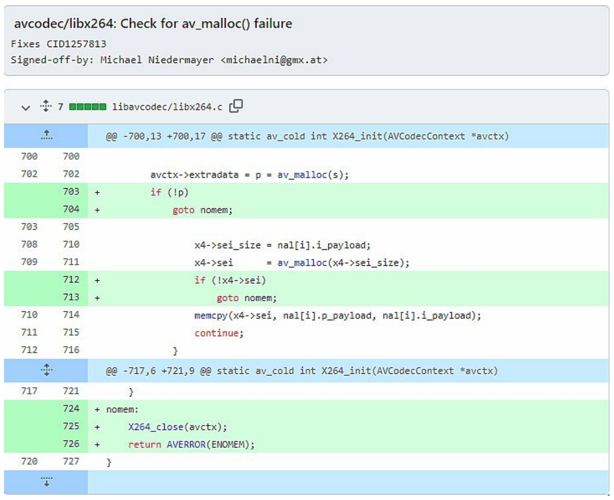
Approach	Extraction	Processing	Inference	Total
Stacking	0.00	0.30	**7.43**	7.73
Commit2vec	10.75	195.33	2.53	208.61
PatchRNN	10.75	0.73	1.18	12.66
SPI	0.00	0.57	1.52	2.09
Transformer	0.00	0.57	1.71	2.28
VulFixMiner	0.00	0.57	1.66	2.23
E-SPI	**10.75**	**199.41**	4.07	**214.23**

Answer to RQ4: Although the accuracy of E-SPI decreases in a real deployment environment where the data distribution is different from the used dataset, the performance of E-SPI is still higher than the baselines. Furthermore, the efficiency of our approach (i.e., the prediction time) is also acceptable compared to the baselines.

5.5 RQ5: A Qualitative Analysis of Results by E-SPI

From the previous experimental results, we can conclude that E-SPI performs well in accuracy and F1. Furthermore, we supplement a qualitative analysis of the results obtained by E-SPI to understand the reason for the correct predictions and misclassifications. Since the prediction results by E-SPI include 1,761 true positive, 1,688 true negative, 242 false positive and 169 false negative samples, we randomly select 100 samples from each class for manual analysis. After the analysis, we discover the following phenomena:

- Compared to those true negative samples (i.e., correctly identified as non-security patches), most of the true positive samples (i.e., correctly identified as security patches) contain security-related words (i.e., vulnerability noun, such as "buffer overflow") in the commit message or have security patterns in the change code. Specifically, 75 out of 100 true positive samples contain obvious security-related words in commit message, such as "fix memory leak", "avoid integer overflow" and "fix division by zero". For example, the commit *ff76335* with commit message "avfilter/zscale: fix memory leak." and the commit *51c1ebb* with commit message "Fix SCSI off-by-one device size.". In addition, 20 true positive samples do not contain obvious security-related words in the commit message, but have security patterns in the change code, such as adding condition check and modifying condition check. For example, the commit *066dc04* adds condition check to avoid null pointers, as shown in Fig. 9.
- For those misclassified samples, we find that they may also be related to security-related words in the commit message. Specifically, some of the false positive samples (i.e., misclassified as security patches) contains security-related words in commit message, just like the true positive samples. For example, as shown in Fig. 10, the non-security patch *9560915* contains "overflow check" in commit message, but is not a security patch. Similarly, some of the false negative samples (i.e., misclassified as non-security patches) do not contain security-related words in commit message. For example, the security patch *01e5e97* with commit message " mjpegbdec: Fix incorrect bitstream buffer size." does not contain any security-related words, but provides obvious security information in the commit message.



avcodec/libx264: Check for av_malloc() failure

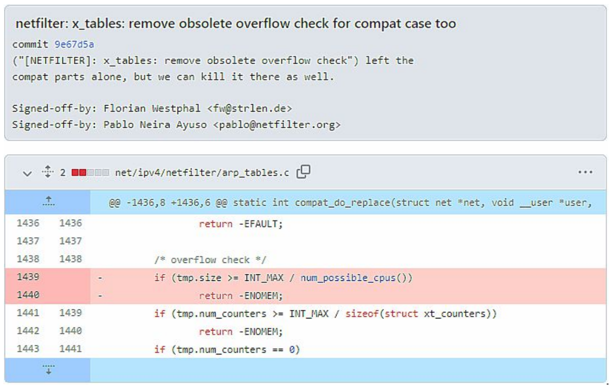
```
Fixes CID1257813
Signed-off-by: Michael Niedermayer <michaelni@gmx.at>
```

```
7 libavcodec/libx264.c
@@ -700,13 +700,17 @@ static av_cold int X264_init(AVCodecContext *avctx)

           avctx->extradata = p = av_malloc(s);
+          if (!p)
+              goto nomem;

                  x4->sei_size = nal[i].i_payload;
                  x4->sei      = av_malloc(x4->sei_size);
+                 if (!x4->sei)
+                     goto nomem;
                  memcpy(x4->sei, nal[i].p_payload, nal[i].i_payload);
                  continue;
              }
@@ -717,6 +721,9 @@ static av_cold int X264_init(AVCodecContext *avctx)
      }
+ nomem:
+     X264_close(avctx);
+     return AVERROR(ENOMEM);
  }
```

Fig. 9: Commit *066dc04*.



netfilter: x_tables: remove obsolete overflow check for compat case too

```
commit 9e67d5a
("[NETFILTER]: x_tables: remove obsolete overflow check") left the
compat parts alone, but we can kill it there as well.

Signed-off-by: Florian Westphal <fw@strlen.de>
Signed-off-by: Pablo Neira Ayuso <pablo@netfilter.org>
```

```
2 net/ipv4/netfilter/arp_tables.c
@@ -1436,8 +1436,6 @@ static int compat_do_replace(struct net *net, void __user *user,
                  return -EFAULT;

              /* overflow check */
-             if (tmp.size >= INT_MAX / num_possible_cpus())
-                 return -ENOMEM;
              if (tmp.num_counters >= INT_MAX / sizeof(struct xt_counters))
                  return -ENOMEM;
              if (tmp.num_counters == 0)
```

Fig. 10: Commit *9560915*.

- In addition, among the misclassified samples, we find that there are two mislabelled samples (i.e., commit *695be0e* and commit *3614364*). Both are security patches, but labeled as non-security patches and E-SPI make the correct predictions for them.

From the phenomena, E-SPI seems to identify most security and non-security patches based on security-related words and security patterns, while also causing some samples to be misclassified. Therefore, we can infer that E-SPI pays close attention to security-related words in commit messages and security patterns in change code when identifying security patches. Furthermore, we also find that E-SPI can identify silent security patches which are mislabeled as non-security patches.

Answer to RQ5: When identifying security patches, E-SPI pays close attention to security-related words in commit messages and security patterns in change code.

6 Discussion

In this section, we first discuss the impact of different numbers of paths (i.e., k) and the ratio of within-changes paths and within-context paths to the performance, and then we conduct an online survey to investigate how developers evaluate the results generated by E-SPI. Finally, we discuss several potential limitations of E-SPI.

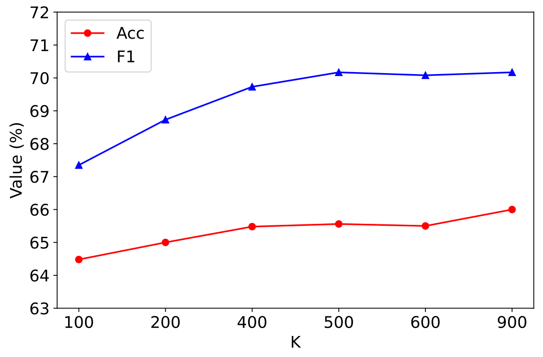


Fig. 11: The impact of k on accuracy and F1.

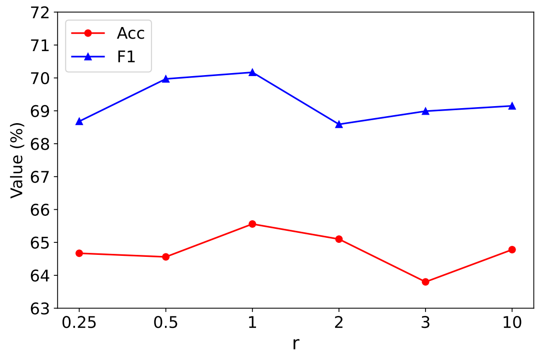


Fig. 12: The impact of r on accuracy and F1.

6.1 Choice of *k*

In Section 4.4, we set the maximum number of AST paths k to 500. We also tune k within a range of 100 to 900 for AST-based code change encoder. Specifically, we turn off the commit message encoder and just use the code change encoder for the experiment to evaluate the effect of the number of paths on the code change encoder, and the other settings are the same as in Section 4.4 for a fair comparison. The experimental results are shown in Fig. 11.

From Fig. 11, we can observe that with increasing number of paths, accuracy and F1 are improved simultaneously, which intuitively reveals that the encoder can learn more effective information with increasing number of paths. After 500 paths, the accuracy and F1 do not improve significantly by further increasing the number of the paths. To balance performance and efficiency (i.e., more paths tend to raise the computation time), we set the AST number to 500 while representing the changed code in our experiments.

6.2 The ratio of within-changes paths and within-context paths

Since the AST paths we extracted from source code include two parts: within-changes and within-context, the ratio of them may effect the final performance. Here, we define the ratio of within-changes paths to within-context paths as r. To study the effect of r on the performance, we keep the maximum number of AST paths k to 500, and conduct experiments with different r ranging from 0.25 to 10. The experiment settings are the same with Section 6.1, and the results can be seen in Fig. 12.

We can see that when r is equal to 1, the accuracy and F1 reach the peak. This means that the number of within-changes paths and within-context paths is the same, resulting in both types of information being fed into the model fairly. However, we can not guarantee that the model achieves the best performance when r is equal to 1, since the relationship between r and performance is not obvious. When using this model, r needs to be tuned based on the dataset.

6.3 Online survey

We conduct an online survey to investigate how well developers trust the results generated by E-SPI.

Dataset. We randomly select 10 samples from the test set and use the model prediction results to conduct the survey analysis.

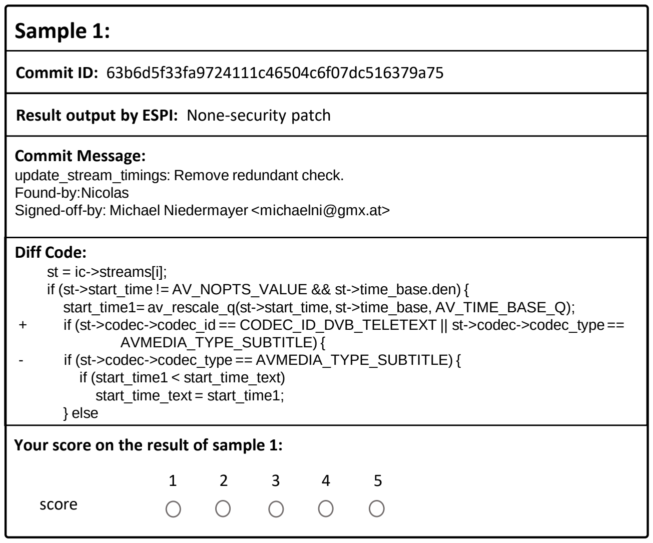
Participant Recruitment. We recruit 30 people from industrial companies and our universities to participate in the online survey. Among the participants, 16 of them come from industry and the rest are from academia. They are all software developers, from PhD students, post-doctoral researchers to software engineers. All of them are not coauthors of our work and use Github for software development with more than 3 years of development experience.

Experiment Procedures. We start the online survey with a brief introduction. We explain to the participants that our task is to assess their level of trust in the results produced by E-SPI. Then the participants are required to provide their personal information relevant to the survey. To quantitatively measure how much they trust the results, we define the rating scale as 1 to 5 where a higher score means that the more confidence the participants have in the results. Commit messages and change code are presented to participants so that they can make their own predictions for each commit. After that, they are required to rate the confidence they have in the results of E-SPI by comparing them with their own prediction for the 10 test samples. They are required to complete the survey online, where Fig. 13 demonstrates part of the questions and the completed survey is available on https://forms.gle/m7SSxgqBYDk3mp9u9.

Survey Results. The average score of each sample is shown in Fig. 14. We can see that the average scores of sample 1, 2, 3, 6 and 10 are above 4, which means that the results of E-SPI are in good agreement with the participants' own predictions. Except for sample 5 and 9, all other sample scores are greater than 3, which means that the results are acceptable for participants. Generally, the average score for all 10 samples is 3.78 which is better than Acceptable and close to Good.

6.4 Limitations

By comparing the results in Section 5.1 and Section 5.5, we find that the decrease in the real deployment is obvious. Although it is a common sense that deep learning-based techniques tend to have the poorer performance on the out-of-domain dataset, which is different from the training data, it actually hinders the model deployed to the real production environment and we also admit it is the limitation



Sample 1:

Commit ID: 63b6d5f33fa9724111c46504c6f07dc516379a75

Result output by ESPI: None-security patch

Commit Message:
update_stream_timings: Remove redundant check.
Found-by:Nicolas
Signed-off-by: Michael Niedermayer <michaelni@gmx.at>

Diff Code:
```
     st = ic->streams[i];
     if (st->start_time != AV_NOPTS_VALUE && st->time_base.den) {
         start_time1= av_rescale_q(st->start_time, st->time_base, AV_TIME_BASE_Q);
+        if (st->codec->codec_id == CODEC_ID_DVB_TELETEXT || st->codec->codec_type ==
                 AVMEDIA_TYPE_SUBTITLE) {
-        if (st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE) {
             if (start_time1 < start_time_text)
                 start_time_text = start_time1;
         } else
```

Your score on the result of sample 1:

	1	2	3	4	5
score	○	○	○	○	○

Fig. 13: Part of the questions in the survey.

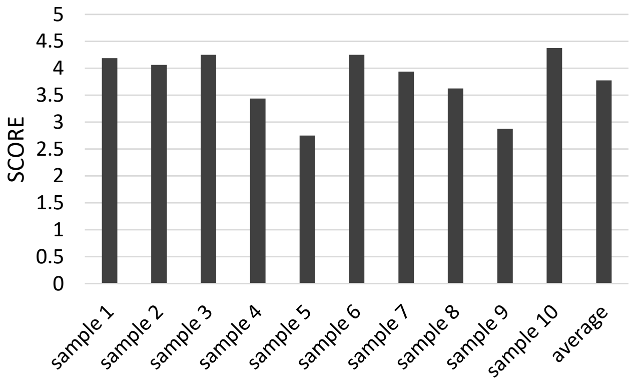


Fig. 14: The average score of all samples in the survey.

of our work. To mitigate this limitation, we propose some underlying solutions to improve the generalizability of the model.

- Improving the diversity of the training data. It is a straightforward idea to expend the dataset. Because the used data in our experiment is quite limited, including more commit data from more projects may improve the performance on unknown data. However, it is difficult in the real scenario since the cost of labeling a large scale of dataset is substantial. Hence, an alternative option is to perform data augmentation [41] on the existing dataset to enrich the diversity. For example, we can perform some semantic-equivalent operators on the commit and transform it into different versions to improve the diversity of data for security patch identification. We leave the design for the semantic-equivalent augmentation on the commit as our future work.
- Improving the generalization ability of the model. In terms of the model design, although the generalization ability of the model is a long-term problem in deep learning community, recent advanced researches on the ensemble learning [42], [43], [44] have proved that the problem of the generalization capability can be mitigated. Hence, we could utilize ensemble learning, which combines multiple learners, to improve the generalization ability for security patch identification.

7 Related Work

In this section, we briefly introduce related works about security patch identification, graph neural networks, and code representation techniques.

7.1 Security Patch Identification

A large amounts of silent security patches in the real world have not been noticed, causing them to be disclosed. Identifying these silent patches can help developers fix security bugs in their software in time and avoid the N-day attack. However, thousands of patches are submitted every day to Github, Bugzilla or other platforms to fix the bugs in their software. It is a great challenge to manually identify security patches in such a large number of patches. Therefore, automatic security patch identification becomes essential. Since the traditional static analysis approaches suffer from high false positive, people turn to learning-based approaches. In the early stages, conventional machine learning algorithms are applied to identify security patches. For example, Zhou et al. [11] propose to utilize a stacking model that assembles six individual classifiers for security patch identification. After that, deep learning-based approaches gradually became mainstream for this task due to their powerful capabilities. Wang et al. [32] propose PatchRNN, which ensembles two BiLSTM models to generate the feature vectors for each patch. Zhou et al. [14] propose SPI, which leverages LSTM and CNN together to learn the representation of each patch. Lozoya et al. [19] propose Commit2vec, which also employs BiLSTM to learn the representation from AST paths. We can also find that, in addition to commit message, people are trying to leverage the code information for security patches identification. In contrast to the approach in the preliminary stage [11] which only considers commit messages as input, Commit2vec starts extracting AST paths from diff code as input, while PatchRNN and SPI both use commit messages along with code for security patch identification. Similar to PatchRNN and SPI, we leverage both commit message and code to identify security patches.

7.2 Graph Neural Networks

In our real world, there are many graph types of data, which is composed of objects and their relationships, such as social networks, molecules, and programs. Before the advent of GNN, due to the great progress of conventional neural networks such as CNN and LSTM, they have been widely applied in various fields such as image classification and natural language processing [8], [45], [46], [47], [48]. However, people gradually find that these conventional neural networks can only operate on regular Euclidean data such as images and texts [49]. Therefore, graph neural networks are proposed to extend neural networks to non-Euclidean domains. In just a few years, many variants of graph neural networks have been proposed, which can be mainly categorized into four groups [50]: recurrent graph neural networks [23], [51], convolutional graph neural networks [52], [25], graph autoencoders [53], [54] and spatial-temporal graph neural networks [55], [56]. In this paper, we apply GGNN, a type of recurrent graph neural network, to the dependency graph of commit messages for graph representation learning.

7.3 Code Representation

To represent code for machine learning, in a preliminary stage, the code is simply treated as a sequence, such as a sequence of tokens or a bag of words. RNN techniques such as LSTM and GRU are used to generate code representations. However, it was discovered that treating code as a sequence lost its structure information, since code are highly-structured data format. These program structures reveal the syntax and semantics of the code [57]. Therefore, many works are trying to use program structures for code representation such as ASTNN [58], ATOM [21], Program Graph [28], Devign [15], VulSniper [59] and HGNN [16]. Specifically, Allamanis et al. [28] uses a graph gated neural network and a program graph to learn function name and detect variable misuse. Devign [15] and Vulsniper [59] employ the code property graph to learn vulnerable functions. ASTNN [58] uses an AST representation to represent source code and perform source-code related tasks, such as clone detection and source code classification. Some other works have also attempted to use low-level programming representations such as LLVM. For example, Flow2Vec [60] employs an interprocedural value-flow graph and LLVM intermediate representation to learn a meaningful representation of the program. Compared with these works, we employ a Bi-LSTM to incorporate AST path information to learn the semantics of patches.

8 Conclusion

In this paper, we propose a well-designed tool E-SPI for automated security patch identification, which fully utilizes the structure information in a commit for effective identification. By the AST-based code change encoder to extract the contextual AST paths about the changed code with BiLSTM to learn its representation and the graph-based commit message encoder to construct the dependency graph for the commit message with the graph neural network to learn the token relations from the message, E-SPI outperforms current state-of-the-art baselines significantly with 4.01% higher accuracy and 4.42% F1 score on the current dataset, increases 6.03% accuracy and 7.38% F1 in the real deployment environment.

9 Acknowledgments

This research is partially supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry of Education, Singapore under its Academic Research Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

[1] (2021) Logging services. [Online]. Available: https://logging.apache.org/log4j/2.x/security.html/

[2] (2021) Logging services. [Online]. Available: https://logging.apache.org/log4j/2.x/

[3] (2021) Nvd. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss/

[4] (2017) Sourceclear. [Online]. Available: https://app.sourceclear.io/

[5] (2018) Sourceclear. [Online]. Available: https://github.blog/2018-11-08-100m-repos/

[6] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[7] T. Ball, "The concept of dynamic analysis," in *Software Engineering—ESEC/FSE'99*. Springer, 1999, pp. 216–234.

[8] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2285–2294.

[9] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[10] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[11] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.

[12] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[13] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[14] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *arXiv preprint arXiv:2105.14565*, 2021.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[16] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=zv-typ1gPxA

[17] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[18] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[19] R. C. Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *SN Computer Science*, vol. 2, no. 3, pp. 1–16, 2021.

[20] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.

[21] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, 2020.

[22] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

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

[24] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.
[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[26] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *arXiv preprint arXiv:1804.00823*, 2018.
[27] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference*, 2019, pp. 417–426.
[28] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.
[29] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.
[30] 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.
[31] S. Liu, X. Xie, L. Ma, J. Siow, and Y. Liu, "Graphsearchnet: Enhancing gnns via capturing global dependency for semantic code search," *arXiv preprint arXiv:2111.02671*, 2021.
[32] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, "Patchrnn: A deep learning-based system for security patch identification," *arXiv preprint arXiv:2108.03358*, 2021.
[33] M.-C. De Marneffe, B. MacCartney, C. D. Manning *et al.*, "Generating typed dependency parses from phrase structure parses." in *Lrec*, vol. 6, 2006, pp. 449–454.
[34] G. Attardi and M. Simi, "Dependency parsing techniques for information extraction," *Dependency Parsing Techniques for Information Extraction*, pp. 9–14, 2014.
[35] Y. Chen, L. Wu, and M. J. Zaki, "Reinforcement learning based graph-to-sequence model for natural question generation," *arXiv preprint arXiv:1908.04942*, 2019.
[36] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.
[37] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *arXiv preprint arXiv:1811.01824*, 2018.
[38] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.
[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
[40] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.
[41] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.
[42] Y.-B. Kim, K. Stratos, and D. Kim, "Domain attention with an ensemble of experts," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 643–653.
[43] J. Guo, D. J. Shah, and R. Barzilay, "Multi-source domain adaptation with mixture of experts," *arXiv preprint arXiv:1809.02256*, 2018.
[44] C. Clark, M. Yatskar, and L. Zettlemoyer, "Don't take the easy way out: ensemble based methods for avoiding known dataset biases," *arXiv preprint arXiv:1909.03683*, 2019.
[45] H. Lee and H. Kwon, "Going deeper with contextual cnn for hyperspectral image classification," *IEEE Transactions on Image Processing*, vol. 26, no. 10, pp. 4843–4855, 2017.
[46] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen, "Medical image classification with convolutional neural network," in *2014 13th international conference on control automation robotics & vision (ICARCV)*. IEEE, 2014, pp. 844–848.
[47] Y. Liu and M. Lapata, "Text summarization with pretrained encoders," *arXiv preprint arXiv:1908.08345*, 2019.
[48] R. Nallapati, B. Zhou, C. Gulcehre, B. Xiang *et al.*, "Abstractive text summarization using sequence-to-sequence rnns and beyond," *arXiv preprint arXiv:1602.06023*, 2016.
[49] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.
[50] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.
[51] H. Dai, Z. Kozareva, B. Dai, A. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *International conference on machine learning*. PMLR, 2018, pp. 1106–1114.
[52] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, pp. 3844–3852, 2016.
[53] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, "Learning deep generative models of graphs," *arXiv preprint arXiv:1803.03324*, 2018.
[54] S. Cao, W. Lu, and Q. Xu, "Deep neural networks for learning graph representations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.
[55] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," in *International Conference on Neural Information Processing*. Springer, 2018, pp. 362–373.
[56] A. Jain, A. R. Zamir, S. Savarese, and A. Saxena, "Structural-rnn: Deep learning on spatio-temporal graphs," in *Proceedings of the ieee conference on computer vision and pattern recognition*, 2016, pp. 5308–5317.
[57] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," *arXiv preprint arXiv:2203.11790*, 2022.
[58] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.
[59] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press, 2019, pp. 4665–4671.
[60] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428301

Bozhi Wu is currently pursuing the Ph.D. degree in computer science at Nanyang Technological University, Singapore. His research interests include malware detection, vulnerability detection and deep learning. He focuses on solving conventional problems in software engineering with AI technology.

Shangqing Liu is a research associate at Nanyang Technological University, Singapore. His Ph.D study is supervised by Prof. Liu Yang from NTU. Previously, he was an intern student at Nanyang Technological University from 2017 to 2018. His research interests include graph neural networks (GNNs), program representation learning.

Ruitao Feng received his B.S. degree from the Tianjin University in 2014 and his Ph.D. degree from the Nanyang Technological University in 2021. He is now a research fellow in Nanyang Technological University since 2021. Previously, he was a research assistant in Nanyang Technological University from 2014 to 2020. His research interests include discovering and solving security problems on mobile platform, IoT system and AI-based cybersecurity system.

Xiaofei Xie received his Ph.D, M.E. and B.E. from Tianjin University. He is currently an assistant professor in Singapore Management University, Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance analysis of deep learning systems. He has published some top tier conference/journal papers relevant to software analysis and won two ACM SIGSOFT Distinguished Paper Awards.

Jingkai Siow is currently pursuing his Ph.D. in computer science at Nanyang Technological University, and holds a B.A. in computer science from the Nanyang Technological University. He is also a research graduate officer at Nanyang Technological University. His area of research involves deep learning, cyber-security, and software engineering. He has more than four years of research experience, software development and data engineering.

Shangwei Lin received his B.S. and Ph.D. degree from the National Chung Cheng University in 2003 and 2010. He worked as post-doctoral researcher at NUS and SUTD from 2011 to 2015. In May 2015, he joined NTU as Assistant Professor. His research interests include formal verification, formal synthesis, embedded system design, cyberphysical systems, security systems, multi-core programming, and component-based object-oriented app frameworks for real-time embedded systems.