Empirical Software Engineering (2022) 27:171
https://doi.org/10.1007/s10664-022-10216-4

SPVF: security property assisted vulnerability fixing via attention-based models

Zhou Zhou[1] · Lili Bo[1,2] · Xiaoxue Wu[1] · Xiaobing Sun[1,2] · Tao Zhang[3] · Bin Li[1] · Jiale Zhang[1] · Sicong Cao[1]

Accepted: 20 July 2022

Abstract
The past few years have witnessed the wide application of machine learning models to fix vulnerabilities automatically. However, existing approaches cannot capture the characteristics of vulnerabilities that are helpful to improve the effectiveness of automated vulnerability fixing. In this paper, we propose a novel approach for automatically fixing vulnerabilities, called SPVF. SPVF captures the security property from the descriptive information about the vulnerability. SPVF is based on the attention mechanism and uses the abstract syntax tree as well as the security properties, integrating them using the pointer generator. The experimental results on two public datasets show that SPVF outperforms the state-of-the-art approaches by 13% for C/C++ and 47% for Python. And SPVF is capable of successfully fixing 153 C/C++ vulnerabilities and 276 Python vulnerabilities.

Keywords Vulnerability fixing · Software security · Attention-based models · Pointer generator

1 Introduction

A vulnerability is an open weakness of the software that can be exploited for one or more threats. The users and vendors suffer a lot from past vulnerabilities like Heartbleed (Durumeric et al. 2014) and shellshock (Delamore and Ko 2015). Fixing the vulnerabilities, on the other hand, is an absolute necessity. Still, it is tedious, error-prone and time-consuming to manually fix the vulnerability (Sun et al. 2019; Cao et al. 2022; Wei et al. 2021).

Automatically fixing vulnerabilities can be seen as a subset of Automated Program Repair (APR), which aims at automatically finding a solution to software bugs without

Communicated by: Jacques Klein

✉ Lili Bo
lilibo@yzu.edu.cn

✉ Xiaobing Sun
sundomore@163.com

Extended author information available on the last page of the article.

human intervention. The traditional ways of solving APR problem rely heavily on hand-crafted rules and are applicable to specific languages. On the contrary, methods using machine learning techniques do not need specific rules tailored for the specific languages. Due to the wealth of data that is available with open-source code repositories, Common Vulnerabilities and Exposures (CVE) (CVE 2021), etc. it is possible to use Neural Machine Translation (NMT) models to automatically fix vulnerabilities. NMT models repair the bugs, by learning infinite fixing patterns from previous good patches instead of a small number of previous rules specified. Therefore, NMT approaches are proposed as likely means of translating the vulnerable code into a correct patch.

Nevertheless, the current APR methods based on NMT models fall short due to the ignorance of vulnerability characteristics. With plenty of researches for analyzing the vulnerabilities (Wei et al. 2021; Cheng et al. 2021; Li et al. 2018; Chakraborty et al. 2020) and public available vulnerability descriptions on exploit database (CWE 2021a) and CVE, it is time to explore the potential of the vulnerabilities characteristics after analyzing. The vulnerability descriptions published by CVE offer professional insight into the characteristics of vulnerabilities. The causes and effects of each vulnerability are carefully described in these descriptions. Moreover, the code snippets of the vulnerabilities are supposed to be with fewer lines of code and fewer logical changes involved (Ni et al. 2020; Li and Paxson 2017; Wei et al. 2021). Hence, the automatic patching model is supposed to perform more successfully by considering these vulnerability characteristics.

There are two challenges in capturing vulnerability characteristics: 1) vulnerability characteristics are hard to be extracted: the descriptions in natural languages of vulnerabilities have lots of noisy information that the security-relevant information cannot be noticed and learned by NMT models. 2) the characteristics of vulnerabilities are difficult to combine with NMT models, i.e., the concatenation of all characteristics might decrease the performance because of its incapability of dealing with long sequences (Yang et al. 2020; Tufano et al. 2018) and the comparatively little useful information over the whole sequence to learn.

To deal with the above challenges, we propose Security Property assisted Vulnerability Fixing (SPVF) which is based on the attention mechanism. We also use the function-level segmentation and Abstract Syntax Tree (AST) to provide a higher level of abstraction and to better capture the code characteristics. Besides, the security properties, which refer to the knowledge we extracted from natural language descriptions of the vulnerabilities and the vulnerability categories, are also considered. Second, we introduce the pointer generator network to integrate the AST representation and the security properties. The pointer generator learns to choose one token in the local context and copy the token as our prediction. The pointer generator network learns better with the additional information including the AST representation and security properties. SPVF is thus capable of making better predictions for patching the vulnerability.

With a vulnerable function location and the related exploit as inputs, SVPF works as follows. First, we capture the security property from the descriptive information about the vulnerability. Then, we feed the security property, serialized AST and the function-level code representation into the model for training. For a public exploit with code and description, SPVF can be used to extract the security property and generate a patch after feeding it into the trained model.

We evaluated SPVF on both C/C++ and Python datasets. The results demonstrate that SPVF has a 13% improvement for fixing C/C++ vulnerabilities and a 47% improvement for fixing Python vulnerabilities compared with the state-of-art approach SeqTrans (Chi et al. 2020).

To sum up, this paper makes the following contributions.

- We propose SPVF for automatically fixing vulnerabilities based on an attention-based model, i.e. Transformer, using pointer generator to integrate the security properties and AST representations.
- We evaluated our approach on two public vulnerability fixing datasets. The experimental results show that SPVF can fix 153 C/C++ vulnerabilities and 276 Python vulnerabilities. To the best of our knowledge, this is the best result reported on vulnerability fixing using attention-based models.

The remainder of this paper is organized as follows. Section 2 presents the motivation; Section 3 introduces the background, including transformer architecture and attention mechanism; Section 4 describes the proposed SPVF model; Section 5 presents the details of experimental setting and implementation; Section 6 discusses the experimental results. Section 7 discusses the threats to validity; Section 8 summarizes the related work and Section 9 concludes this work with future research directions.

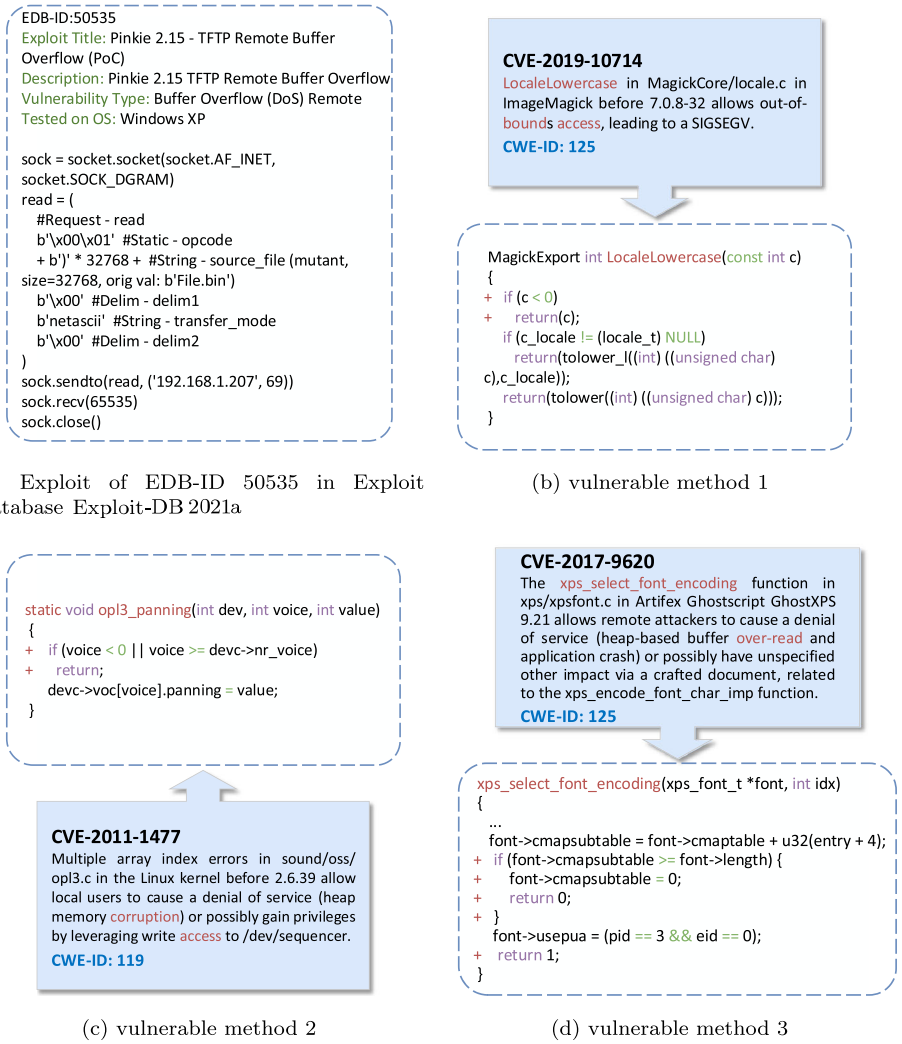
2 Motivation

When a potential vulnerability is discovered, an exploit might be published before the CVE is published. A public repository like Exploit Database (CWE 2021a) shows that among the 42,450 public available exploits, 80% of the exploits are published before the CVEs are published (Palo Alto Networks 2021). Also, the Exploit Database provides reports in natural language with the corresponding vulnerable software in the CVE format. As is shown in Fig. 1a, it is an exploit for *Buffer overflow (DoS) Remote* public on Exploit Database. The Exploit Title, Description and the Vulnerability Type provide vital information for patching the vulnerability. This inspires us to use the natural language descriptions for better patching. Also, note that the exploit database is a CVE compliant archive and that some of the CVE entity has the link to the exploit database with similar descriptions for the vulnerability (Exploit-DB 2021a; Zhou et al. 2021).

Figure 1 also presents three vulnerable methods in CVE. As is shown in Fig. 1b, the words *bound*, *LocaleLowercase* indicate its location and methods to exploit. The category of this vulnerability is *CWE-125*, which is an out-of-bounds read, carrying potentially useful information. As is shown in Fig. 1c, the word "access" indicates that the bounds of the function arguments may be paid more attention to. The category of the vulnerability in Fig. 1d is also *CWE-125*. The word "over-read" indicates missing of boundary check.

From the natural language description with different vulnerable methods in Fig. 1b and c, we can find that they are all about inappropriate access and some boundary issues. The patches for the two methods are both adding the if statement for the boundary check. This inspires us that different vulnerabilities with similar keywords in the natural language descriptions might have similar ways of patching. In addition, from Fig. 1b and d, we can see that with the same CWE category, an if statement for boundaries check is added for fixing the hole even when the natural language descriptions have few keywords in common. This also inspires us that the vulnerable code with the same CWE classification might also have similar ways of patching.

For a newly published exploit on exploit database, the above observations inspire us to first use tools (e.g. BGNN4VD (Cao et al. 2021), ReVeal (Chakraborty et al. 2020), IVDetect (Li et al. 2021)) to locate the exact vulnerable function and then use the exploit with the description and the code at the function level as the input, finally extract the



(a) Exploit of EDB-ID 50535 in Exploit Database Exploit-DB 2021a

(b) vulnerable method 1

(c) vulnerable method 2

(d) vulnerable method 3

Fig. 1 The Exploits and the vulnerable method

security-relevant words as well as the function name and variable name from the description for model training and model inference. Our insight is that, the addition of these security properties not only assists the repair in the way of fixing (e.g., add, delete or modify) but also in emphasizing the most relevant code vocabularies.

Furthermore, notice that the context as well as the structure of the code matter with the vulnerable code (Chen et al. 2021; Lutellier et al. 2020). For instance, if the argument *c* in the *LocaleLowercase* function in Fig. 1b is not captured by the model, there would be less likely for SPVF to successfully patch the vulnerability. So are the argument *voice* in Fig. 1c and the argument *font* in Fig. 1d. The abstract syntax tree captures the code structure. Directly using AST to generate the fixed code snippet is a much too ambitious goal, yet using AST for assistance is a feasible approach for the model to understand the structure of

the code. Based on the above observation, we focus our attention on functional-level repair and use the AST representation for further assistance.

3 Background

SPVF receives vulnerable code at the function level as the input and produces repaired code as the output, which is quite similar to the idea for NMT, that is to translate from one language to another. Because SPVF is based on *Transformer* (Vaswani et al. 2017), a typical architecture for attention-based NMT, we introduce attention-based NMT with special attention to *Transformer*.

An NMT architecture basically consists of an encoding component and a decoding component. After a sequence of tokens is read in by the encoder, the decoder generates an output sequence. In the case of natural language translation, the input sequence of tokens *I love apple* in English translates into the *J'aime la pomme* in French.

3.1 Transformer Architecture

Transformer (Vaswani et al. 2017) uses the standard encoder-decoder NMT architecture. Inside an encoder component, there might be several stacks of encoders with the same architecture. An encoder consists of a multi-head attention layer along with a feed forward attention layer. The inputs of the encoder flow through the multi-head attention layer into the feed forward layer. A decoder also has these two layers. And there is an encoder-decoder multi-head layer between the two layers that draws relevant information from the encoders (Transformer 2020).

3.2 Attention Mechanism

The attention mechanism is proposed by Bahdanau et al. (2015) to enable a neural model to inspect the relevance between each pair of tokens in long sequences. The attention mechanism is an effective way of alleviating the long-dependency problem for long sequences. And the multi-head attention proposed by the *Transformer* is a way of improving the performance of the original attention function. The attention function is computed by the query matrix (Q) generated from the target sequence, the key matrix (K) for the decoder and the value matrix (V) generated from the source sequence as follows:

$$\text{Attention}\,(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

And when it comes to the multi-head attention, there are multiple sets of query, key, value weight matrices.

Transformer uses the multi-head attention in three ways. The first is in the encoder component; the second is between the encoder-decoder component; the third is in the decoder component. For the first and third way of using the multi-head attention, the source sequences are similar to the target sequences, that is the query, key, value matrices are the same. For the second way, the source sequences come from the output of the encoder while the target sequences come from the previous decoder.

4 Proposed Model

4.1 Overview

Figure 2 shows an overview of the SPVF. There are two main stages of SPVF, namely the training stage and the inference stage. In the training phase, we represent the input by the source code, the AST representation and extract the security properties from the raw security-relevant descriptions in natural language. Then, based on the traditional Transformer architecture, we encode the vulnerable code, AST representation and the security properties as the input, the repaired code as the output respectively. Finally, we feed them into the pointer generator network which is an extension of the Transformer, train the model and tune the network with different sets of hyperparameters. In the inference stage, a vulnerable functional-level code with its corresponding security-relevant descriptions feeds into SPVF as the input. SPVF tokenizes the input and then feeds them into the model that has been trained in the training stages, generating a list of top-k patches. Finally, SPVF filters the patches list via compiling.

4.2 Input Representation

The inputs of training model are divided into three parts: 1) the code for vulnerable method 2) the AST representation 3) the security proprieties extracted from the natural language description as well as the CWE category.

4.2.1 Source Code for the Vulnerable Method

We clean the source code before feeding it into the neural network. Concretely, we remove non-ASCII characters, all the comments, remove the content in print function, remove multi-blanks and lines and try to make sure all the split characters like ";", "{", "}" are in the format $[blank][splitcharacter][blank]$.

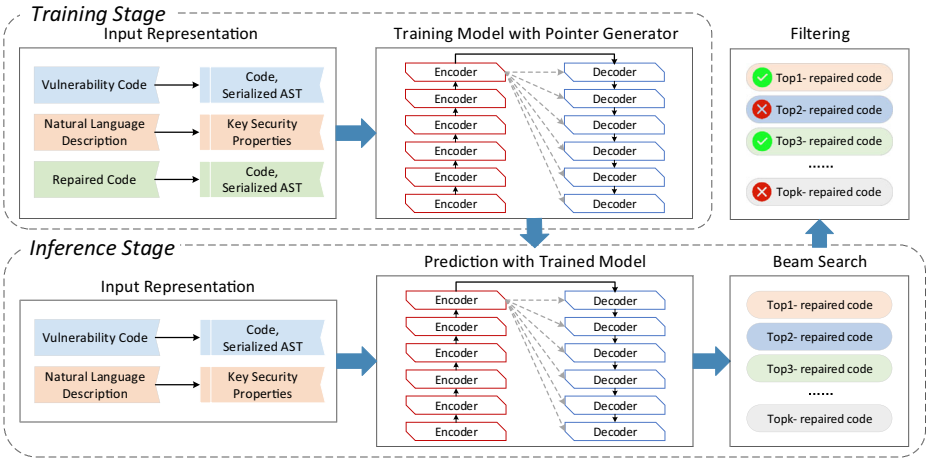


Fig. 2 SPVF's overview

4.2.2 Serialized AST Representation

Figure 3 shows an AST with its corresponding vulnerable code snippet which is first introduced in Fig. 1c. We use the Joern (Yamaguchi) for representing the AST for C/C++ code snippet which is a fuzzy parser that enables the parsing even for a very short code snippet.

As is shown in Fig. 3, AST representation captures the structure of the code. In the AST encoder, SPVF traverses the tree in a depth-first pre-order manner. And when a node is visited, the corresponding node type is recorded, represented as:

$$AST_j = (Node_1, Type_1), ...(Node_i, Type_i) \tag{2}$$

where j denotes the jth function.

4.2.3 Security Property Extraction

The illustrated examples in Section 2 give concrete examples of raw information for the security properties. The information is gathered from *the CVE summary* from the CVE (CVE 2021). However, the raw information is noisy, making it difficult for the model to capture the security-relevant information from the noisy data. We use the NLTK (Bird and Loper 2004) toolkit to preprocess the reports in natural languages, perform lemmatization and remove the stopwords along with HTML tags. TF-IDF algorithm is used to find the most important and relevant information about the vulnerability. TF-IDF algorithm has been widely used for extracting important information in bug reports (Koyuncu et al. 2020; Xin and Reiss 2017; Pradel et al. 2020; Cooper et al. 2021) and it is effective for extracting the security-relevant words from the CVE summary.

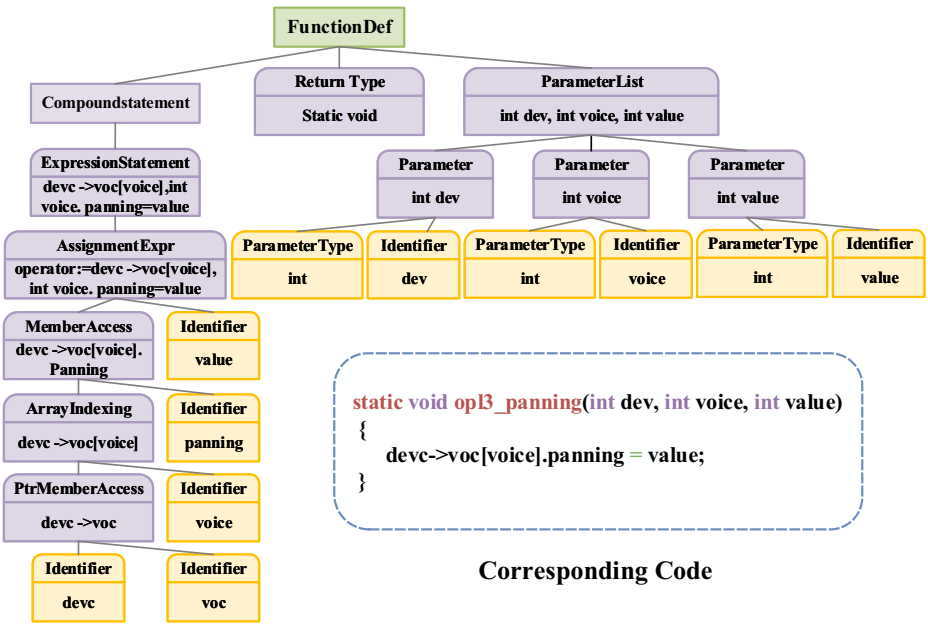


Fig. 3 an illustrated AST for the code snippet

The TF-IDF algorithm is based on the idea that the importance of a word is proportional to the number of times it appears in the entire text database K, and inversely proportional to how common the word is. By calculating the multiplication of the two indicators (i.e. Term Frequency (tf) and Inverse Document Frequency (idf)), we can automatically extract document keywords. Finally, we come to a dictionary with 100 security-relevant words, some of the most important words we consider are "bound", "auth".

Security property refers to the knowledge we extracted from natural language descriptions of the vulnerabilities and the vulnerability categories. Note that the function name and variable name that appear in the natural language descriptions also are taken into consideration. At last, we arrive at the security property consisting of security-relevant keywords, the function name, the variable name and the vulnerability category.

4.3 Model Architecture

Our architecture is based on the "Transformer" and contains encoder components and decoder componeds. The main difference is the way of calculating the attention distributions for code along with its AST representation and security properties.

4.3.1 Encoder

We follow the "Transformer" architecture with six stacks of encoders. The encoder has three inputs, namely the code, serialized AST and the key security properties. To incorporate the order of the words into our model, we add the positional encoding in the first place. The output encoding after positional encoding is passed into the first encoder. Then, for every encoder, the input of the encoder first flows into a multi-head self attention layer. After addition and normalization of the self attention, it then flows into a feed forward layer. The output encoding is then passed to the next encoder.

4.3.2 Decoder

The decoder component also consists of six stacks of decoders. The decoder also has three inputs, namely the code, serialized AST and the key security properties. The layer of encoder-decoder multi-head attention receives the input encoding from the self multi-head attention layer. Also, as can be seen in Fig. 4, the attention distributions directly calculated from the source input and security properties are also part of the input of the multi-head attention layer in the encoder. The remaining parts are the same as the encoder.

4.3.3 Attention for Code and Security Properties

The tokens from the inputs w_i, the source code, the AST representation part and the security properties part are fed into the encoder in the same way, coming to a sequence of encoder hidden states h_i. Then, we calculate the attention distributions according to Bahdanau et al. (2015):

$$e_i^t = v^T \tanh(W_h h_i + W_s s_t + b_{\text{attn}}) \tag{3}$$

$$a^t = \text{softmax}(e^t) \tag{4}$$

where t means on the t's step, s_t is the decoder state. And v, W_h, W_s and b_{attn} are learnable parameters. The attention distributions tell the decoder where to look up by generating a probability distribution over all the input tokens. Then, a weighted sum of hidden states by

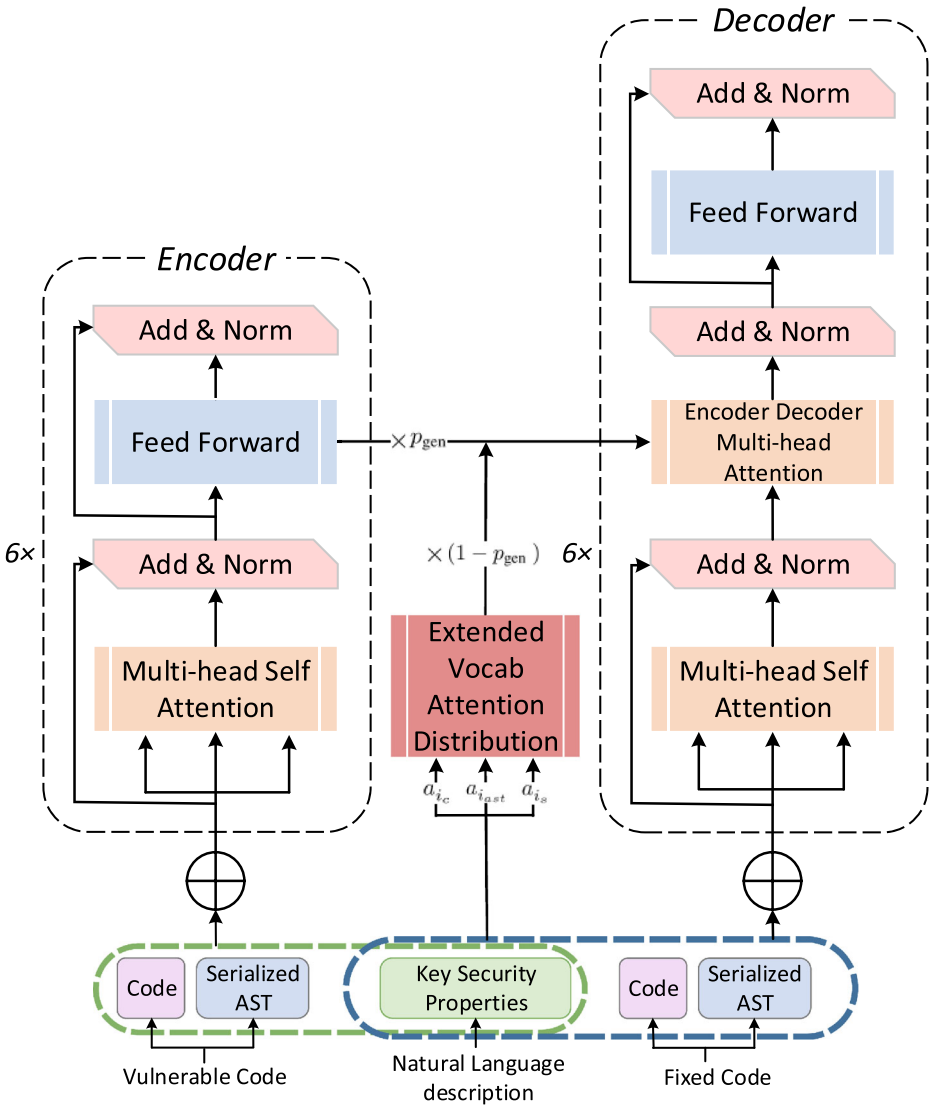


Fig. 4 The architecture used in SPVF

the source code along with AST representation and the security properties are calculated using the respective attention distributions, called the context vector h_t^*:

$$h_t^* = \sum_i a_{i_c}^t h_{i_c} + \sum_i a_{i_{ast}}^t h_{i_{ast}} + \sum_i a_{i_s}^t h_{i_s} \tag{5}$$

where c represents the code and s represents the extracted security properties.

4.3.4 Pointer Generator Network

Proposed by See et al. (2017), pointer generator network was originally used for abstract text summarization. It is proposed due to the basic errors that seq2seq models frequently swamp into. Integration of pointer generator network in SPVF tries to drag the network out of the swamp of not knowing the basics of code like missing the semicolon and the characteristics of the vulnerabilities.

The allowance of both copying words via the source code, the indication of security properties and the generating words from the fixed vocabulary is made possible via a probability. And we use the same name as proposed in Abigail See (See et al. 2017) for p_{gen}. p_{gen} has a value between 0 and 1. The closer is p_{gen} to 0, the more likely for the model to refer to the source code and security properties. Specifically, on every timestep t, p_{gen} is calculated by the context vector h_t^* as well as the decoder states s_t and the decoder input x_t:

$$p_{\text{gen}} = \sigma(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}}) \tag{6}$$

where vectors w_{h^*}, w_s, w_x and scalar b_{ptr} are learnable parameters and σ is the sigmoid function. For any generated tokens, it can be generated via choosing from the extended vocabulary or via sampling from the attention distributions. The extended vocabulary contains all tokens in the code and its security properties in the dataset while the vocabulary only contains a limited number of tokens. So, we come to the final probability distribution over the extended vocabulary:

$$\begin{aligned} P(w) \;=\; & p_{\text{gen}} \times P_{\text{vocab}}(w) + (1 - p_{\text{gen}}) \\ & \times \left(\sum_{i:i=i_s, w_i=w} a_{i_s}^t + \sum_{i:i=i_p, w_i=w} a_{i_c}^t + \sum_{i:i=i_{ast}, w_i=w} a_{i_{ast}}^t \right) \end{aligned} \tag{7}$$

When w is an out-of-vocabulary (OOV) word, the final probability distribution is no longer zero because it can still refer to the a_{i_c}, $a_{i_{ast}}$ or a_{i_s} depending on whether it is code, AST representation or security properties. Thus, we are capable of producing OOV words by copying words via pointing.

4.3.5 Loss Function and Beam Search

The loss function is used to optimize the model. For time step t, the negative log probability of the target word w_t^* is the loss. And the probability is the final probability by taking the extended vocabulary into consideration.

$$\text{loss}_t = -\log P(w_t^*) \tag{8}$$

and the overall loss for the whole sequence is:

$$\text{loss} = \frac{1}{T} \sum_{t=0}^{T} \text{loss}_t \tag{9}$$

On each step, our model selects and holds to B number of the best alternatives instead of merely choosing the word with the highest probability. This is the *Beam search* (Freitag and Al-Onaizan 2017) technique and B is called the beam width. By setting the parameter K in our model, we are capable of getting the top K patches for the vulnerable function instead of the most likely patch.

4.4 Filtering

We filter the patches following the simple rule of whether they can be compiled or not. We do not use the test suite for filtering which APR methods uses, because the test suite for the dataset might be hard to gather, but also the vulnerable code in nature is different from the buggy code where not all vulnerable code snippets have a test case to trigger. In other words, some of the vulnerabilities are about the problem of usage of the potentially vulnerable function whilst no test case can be found to trigger the problem.

5 Experiments

5.1 Research Questions

- RQ1: How does SPVF perform against state-of-the-art automatic vulnerability repair techniques?
  RQ1 is to investigate the SPVF capability for fixing vulnerabilities against the state-of-the-art approaches.
- RQ2: What are the contributions of different components of SPVF?
  RQ2 is to find out whether each component of the model contributes to the overall performance of SPVF and which component has the biggest contribution.
- RQ3: What category of vulnerability is SPVF good at and not good at patching?
  RQ3 aims to understand the SPVF's way of fixing the vulnerabilities by investigating into the category which successful patches belong to and the cases that SPVF failed at patching.

5.2 Dataset

The detailed statistics of these datasets are listed in Table 1. Our experiment is conducted based on two public datasets: the C/C++ vulnerability dataset (Fan et al. 2020) and the Python vulnerability dataset (Wartschinski 2019). The C/C++ vulnerability dataset is from MSR 20 with 188636 C/C++ function pairs. It is scraped from *GitHub* (Github 2007), CVE (CVE 2021) and CWE (Exploit-DB 2021b). The C/C++ dataset is extracted from 348 projects' code repositories with a time span of 17 years, from 2002 to 2019. Despite the abundant information it provides, the vulnerable code that can be used for training is relatively scarce due to the following two reasons. First, a large proportion of the functions are labeled as not vulnerable. This proportion of functions might be useful for tasks like vulnerability detection (Li et al. 2022). But it cannot provide code change information for the vulnerability fixing tasks. Second, many functions are not sliced in an accurate way, failing to be compiled.

To address these issues, we exclude the functions that are labeled as not vulnerable and the functions that fail to be analyzed by the compiler. The rest of the data makes up a dataset containing 10,739 function pairs. The Python dataset was originally used by VUDENC (Vudenc 2021) which is a tool for vulnerability detection. The data is now available on zenodo.[1] After mining the data from GitHub (Github 2007), they collect the vulnerable function by identifying whether it is a vulnerability by capturing keywords like "SQL injection issue"

[1] https://zenodo.org/record/3559203#.XeRoytVG2Hs

Table 1 Details of vulnerability dataset

Statistics	C/C++	Python
Train	8591	3624
Valid	1074	453
Test	1074	453
Total	10739	4530
Avg. code token length	281	55
Avg. security property token length	82	53
Max. code token length	583	561
Max. security property token length	119	136

from the commit. As is described in Section 4.2.3, security properties comprise security-relevant keywords, the function name and variable name that appear in the natural language descriptions and the vulnerability category. For the C/C++ dataset, we extract the security-relevant keywords from the CVE summary and the vulnerability category according to the CWE. For the Python dataset, we directly use the security keywords entity and vulnerability category in the dataset. The dataset with security property is available on zenodo.[2] As can be seen in Table 1, the Python dataset is smaller than C/C++ dataset with a code length. After observing the dataset, we also observe that the Python dataset generally has a more precise and accurate slicing of the code than the C/C++ dataset. For both the C/C++ vulnerability dataset and the Python dataset, we split the data for 80% training, 10% validation, and 10% test.

5.3 Evaluation Metrics

To measure the performance of our approach, we calculated the accuracy for SPVF predicting a successful patch. We introduce the concept of the **compilable patch** and the **correct patch** (Lutellier et al. 2020). Compilable patch refers to the patch that can be compiled successfully. Correct patch refers to the patch that is the same as the target code. When generating K patches by beam search, we deem that the source code is successfully patched if any of the patches out of K is a correct patch.

We use the seemingly coarse way of evaluation for the following two reasons. First, unlike bugs that are mostly related to a failed test suite, some of the vulnerable code snippets do not correspond to a test suite. So, the traditional way of validation via testing is unfeasible for the vulnerabilities. Second, the datasets available for the vulnerabilities are small considering the huge demand for data for training an attention-based model like SPVF, let alone the test suites for some of the vulnerabilities which are even more scarce.

5.4 Parameter Settings

Vocabulary We considered a vocabulary of 8000 tokens and extended it with 1000 input position markers. Source position markers are for *Out Of Vocabulary* tokens so that the identities can be recovered after generation. Concretely, the token not in the vocabulary is represented by numbered $\langle unk - i \rangle$. For example, any $\langle unk \rangle$ token in the text is replaced

[2] https://zenodo.org/record/6324846

with $\langle unk-1 \rangle$ if it appears in the first input position, $\langle unk-2 \rangle$ if it appears in the second input position, and so on. By testing a number of settings, we find that 8000 tokens and 1000 position markers are able to represent over 99.6% of the tokens.

Network Parameters We used the grid search to tune the hyperparameters. After training and evaluating each possible hyperparameters sets, we select the hyperparameter that produces the best results. We set the peak learning rate of 0.08 and used the *inverse square root* as a learning rate scheduler. It sets a constant learning rate for the first k, then exponentially decays the learning rate until pre-training is over. And we set the warm up for 500 parameter updates. We used the Adam (Kingma and Ba 2015) as the optimizer. For each training, we stopped at coverage or until we reached 80 epochs. The primary parameters are shown as follows:

- Peak learning rate: 0.08
- Optimizer: adam
- Dropout: 0.1
- Attention dropout: 0.1

Input and Output The input to the SPVF as illustrated in Fig. 4 has two parts, the natural language description part and the function-level vulnerable code part. The output is the code patches.

Usage We are capable of using the SPVF after its being trained to fix a vulnerability. A list of repaired code patches shall be provided to give the user suggestions of fixing the vulnerability.

5.5 Baselines

We consider **SeqTrans** (Chi et al. 2020) as our baseline which is a vulnerability tool using NMT techniques and can be applied for C/C++ and Python dataset. It is a state-of-the-art tool for vulnerability fixing using NMT models. It is a tool for automatic vulnerability fix via sequence to sequence learning. It leverages data flow dependencies and uses the "Transformer" model.

5.6 Methodology

5.6.1 RQ1 Implementation

To answer RQ1, we first obtained the security property (vulnerability category and the security-relevant words) and the AST representation. We classified C/C++ vulnerabilities according to the classification criteria by CWE (CWE 2021b). According to CWE, The classification is roughly aligned with MITRE's research into vulnerability theory and there is minimal overlap between different categories. For the Python dataset, we directly adopted the vulnerability category used in the dataset. Classification details[3] are shown in Table 2.

Because the code for SeqTrans hasn't yet been published, we strictly followed the steps of SeqTrans. Instead of using AST representation to assist in fixing, SeqTrans uses the data

[3]Note that a small number of vulnerabilities cannot be classified into the classifications listed due to the absence of CWE id information in the dataset.

Table 2 Details of category for vulnerability dataset

Class	Category
C/C++	
0	Improper access control (284)
1	Improper interaction between multiple correctly-behaving entities (435)
2	Improper control of a resource through its lifetime (664)
3	Incorrect calculation (682)
4	Insufficient control flow management (691)
5	Protection mechanism failure (693)
6	Incorrect comparison (697)
7	Improper check or handling of exceptional conditions (703)
8	Improper neutralization (707)
9	Improper adherence to coding standards (710)
Python	
0	SQL
1	Cross site request forgery (XSRF)
2	Command injection
3	Open redirect
4	Remote code execution
5	Cross site scripting (XSS)
6	Path disclosure

flow dependencies to assist fixing. For C/C++ vulnerability dataset, we also used the Joern (Yamaguchi) for the construction of data flow dependencies. For Python dataset, we used the open source tool on GitHub (Python 2021).

5.6.2 RQ2 Implementation

To answer RQ2, we performed a series of ablation tests to understand the impact of each component. For all the experiments, we used beam size = 10. We removed each component in turn. We first removed the pointer generator from SPVF. We directly used the "Transformer" model with the same hyperparameter with the concatenation of the source code and its corresponding security properties. Then, to better understand the contribution of the security properties, we further conducted the experiences only with "Transformer". We used the same dictionary for C/C++ dataset and Python dataset with 100 most security-relevant words. In addition, we conducted several case studies to observe the function of security property as well as the pointer generator.

5.6.3 RQ3 Implementation

To answer RQ3, we firstly analyzed the categories of vulnerabilities that are frequently fixed by SPVF. Secondly, we conducted several case studies and investigated the fixes that are particularly good and poor by the category. By analyzing the data and observing the actual cases, we try to understand the way that SPVF fixes the vulnerabilities.

6 Results

6.1 Effectiveness of SPVF (RQ1)

As shown in Table 3, the results are displayed as x/y, where x represents the number of vulnerabilities correctly fixed and y represents the number of vulnerabilities with compilable patches. The results show that SPVF achieves the best performance on both the C/C++ and Python datasets. On the C/C++ dataset, when beam = 1, SPVF fixes 20 and 22 more vulnerabilities compared with SeqTrans and Transformer, respectively, that is, a 13% increase compared with the state-of-the-art approaches. When beam = 10, SPVF fixes 18 and 27 more vulnerabilities compared with SeqTrans and Transformer, respectively. On the Python dataset, there is a significant improvement compared with basic Transformer and SeqTrans with over 70 more vulnerabilities successfully fixed by SPVF. In particular, SPVF outperforms SeqTrans by 47% on the test set when beam = 10. And the patches are available on GitHub.[4]

The result also demonstrates that SPVF obtains better performance on Python dataset than on C/C++ dataset. The results might be due to the following two reasons. First, as is shown in Table 1, the average length of the Python code is shorter than the C/C++ code. And a shorter length of input tokens enables SPVF to generate the successful patches more easily. Second, Python is a scripting language that generally has a relatively simple syntax and semantics (Wikipedia 2021). It might be easier for the NMT models to learn the fixing patterns, thus yielding higher performance. C/C++ dataset, on the other hand, contains longer sequences and more complex syntax and semantics.

Both SeqTrans and SPVF are based on Transformer architecture. And SeqTrans and SPVF use the representation of the source code. The ignorance of security properties and the difference in integrating the properties into the model might contribute to the reduced effectiveness of SeqTrans.

> **Result 1**: SPVF outperforms state-of-the-art approaches by **13%** for C/C++ and **47%** for Python. SPVF fixes **153** C/C++ vulnerabilities and **276** Python vulnerabilities on the test dataset.

6.2 The Contribution of Each Component

As shown in Table 4, we can see that both the security property and the pointer generator significantly contribute to SPVF because the performance substantially degrades when training out of them. For the C/C++ vulnerability test set, there is a 12% improvement when security property is added to the model and a cumulative 16% improvement when pointer generator is introduced to integrate the properties. For the Python vulnerability test set, a significant 30% improvement is witnessed when the security property is added and a cumulative 64% improvement is observed when the pointer generator is introduced.

Both the security property and the pointer generator are more effective for the Python dataset, which may be because the Python dataset contains less noise than the C/C++ dataset. We also investigated the actual property security extracted, and we found that Python security properties are limited to specific words and generally have more indications

[4] https://github.com/SPVF/SPVF-for-vulnerability-fixing

Table 3 Comparison with the state-of-the-art baseline approaches

Approach	Beam	C/C++ vul. dataset		Python vul. dataset	
		Valid	Test	Valid	Test
Transformer	1	123/696(11.4%)	113/753(10.5%)	157/431(34.6%)	157/441(34.6%)
	10	129/702(12.0%)	131/753(12.1%)	170/434(37.5%)	168/444(37.0%)
SeqTrans	1	125/700(11.6%)	122/755(11.3%)	185/442(40.8%)	176/440(38.8%)
	10	141/723(13.1%)	135/758(12.5%)	193/443(42.6%)	187/441(41.3%)
SPVF	1	145/832(13.5%)	140/846(13.0%)	247/442(54.5%)	244/445(53.8%)
	10	**159/843**(14.8%)	**153/851**(14.2%)	**280/450**(61.8%)	**276/451**(60.9%)

The results are displayed as x/y, with x the number of bugs correctly fixed and y the number of bugs with compilable patches. The highest score for each benchmark is in bold

like "add" and "remove". Many of the C/C++ vulnerability codes require major changes of the code to fix. By contrast, most of the successful patches only consist of minor changes. The major changes of the code are obviously out of the capability of SPVF.

Figure 5 is a Python code snippet that cannot be fixed by Transformer. When the security properties are added, the vulnerability, the code snippet can be fixed. It can be noticed that the security property contains the words "ssh" and "command injection", which are highly relevant to the code snippet. The program might then pay more attention to the "ssh" command resulting in the successful patching.

Figure 6 shows a C/C++ code snippet that cannot be fixed until the pointer generator is introduced. Note that the variable name *EntrySyncCallbackHelper* is an out of vocabulary word that cannot be generated by predicting using the vocabulary. Models with Pointer generator successfully patch this vulnerability by generating via pointing, that is to refer to the class name *EntrySyncCallbackHelper* in the same sentence.

Result 2: Both the security property and the pointer generator in our approach contribute to the overall performance of SPVF.

6.3 Investigation into SPVF's Patches (RQ3)

Figure 7 shows the number of correct patches over categories in the C/C++ dataset with the value on the horizontal axis corresponding to the category defined in Table 2. We can

Table 4 The result for the contribution of different Components of SPVF

Model	C/C++ vul		Python vul	
	Valid	Test	Valid	Test
Transformer	129/702	131/753	170/434	168/444
Transformer+AST+Security Property	149/822	147/788	253/443	219/441
Transformer+AST+Security Property+Pointer Generator (SPVF)	159/843	153/851	280/450	276/451

```
command injection, ssh
```

```
- ssh_cmd = 'svcinfo lsnode -delim ! %s' % node['id']""]
+ ssh_cmd = ['svcinfo', 'lsnode', '-delim', '!', node['id']]""]
```

Fig. 5 Example of vulnerability only fixed by adding the security property(Python)

observe some conclusions in Fig. 7 that SPVF excels at fixing *Improper Access Control* and *Incorrect Calculation* in C/C++ dataset, *Command injection* and *Cross Site Request Forgery(XSRF)* in Python dataset. In addition, Fig. 8 shows the category that the number of correct patches over categories in Python dataset with the value on the horizontal axis corresponds to the category defined in Table 2.

Case Study: Improper Check or Handling of Exceptional Conditions Figure 9 shows a case of CWE-476, that is *NULL Pointer Dereference* belonging to category 7, which belongs to category 7, that is *Improper Check or Handling of Exceptional Conditions*. From Fig. 7, we find that *Improper Check or Handling of Exceptional Conditions* is a category that SPVF is not good at fixing. And the vulnerability needs the addition of line $cdef \rightarrow ents = 0$. SPVF only manages to generate the result $\langle unk \rangle = \langle unk \rangle$. It can be seen that SPVF has already learned the patterns of fixing but failed due to uncertainty of the variable name as well as the value that needs to be assigned to the variable name. The poor performance of the vulnerability fixing in this category might be the too many possibilities for the variable name.

Case Study: Improper Control of a Resource Through its Lifetime The vulnerability in Fig. 10 belongs to CWE-119, which is Improper Restriction of Operations within the Bounds of a Memory Buffer - (119) and belongs to *Improper Control of a Resource Through its Lifetime* which is a category that SPVF especially excels at fixing for C/C++ vulnerability dataset. After changing the return function by returning another function *adoptRefWillBeNoop* instead of directly returning an object, the code is no longer vulnerable. It can also be noticed that SPVF successfully predicts the variable name which is not in the vocabulary via pointing to the source code. And the security properties "bound", "CreateFileResult" also give indications about the way of fixing it.

Case Study: Command Injection As is shown in the previous Fig. 5 SPVF excels at fixing the Python code under the *Command injection* category. We observe some other examples

```
Class-2, bound, allow
```

```
EntrySync* EntrySync::copyTo(DirectoryEntrySync* parent, const
String& name, ExceptionState& exceptionState) const{
-   RefPtr helper = EntrySyncCallbackHelper::create();
+   EntrySyncCallbackHelper* helper =
+            EntrySyncCallbackHelper::create();
     m_fileSystem->copy(this, parent, name, helper-
>successCallback(), helper->errorCallback(),
DOMFileSystemBase::Synchronous);
     return helper->getResult(exceptionState);
 }
```

Fig. 6 Example of vulnerability only fixed by integrating with pointer generator (C/C++)

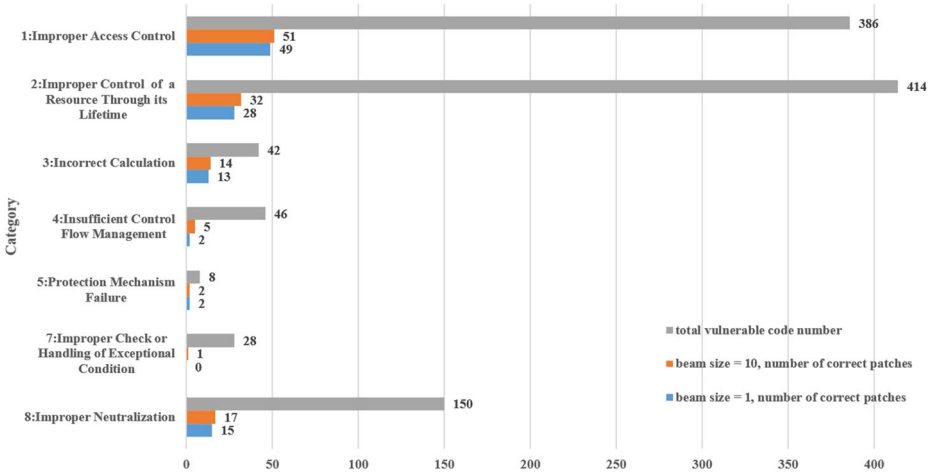


Fig. 7 The number of *correct patch* over categories (C/C++), value on the horizontal axis corresponds to the category defined in Table 2

with similar fixing patterns by removing %s in the code and eliminating the potential hole that exists in the code. The category Command injection frequently successfully patched probably because of the relatively common pattern of fixing.

Case Study: Example of Category SQL Being Patched Figure 11 is a vulnerability that needs to change the content of URL. SPVF performs poorly at fixing Python code under the *SQL* category. The reason that Python patches this vulnerability successfully probability because of the minor change this case involves.

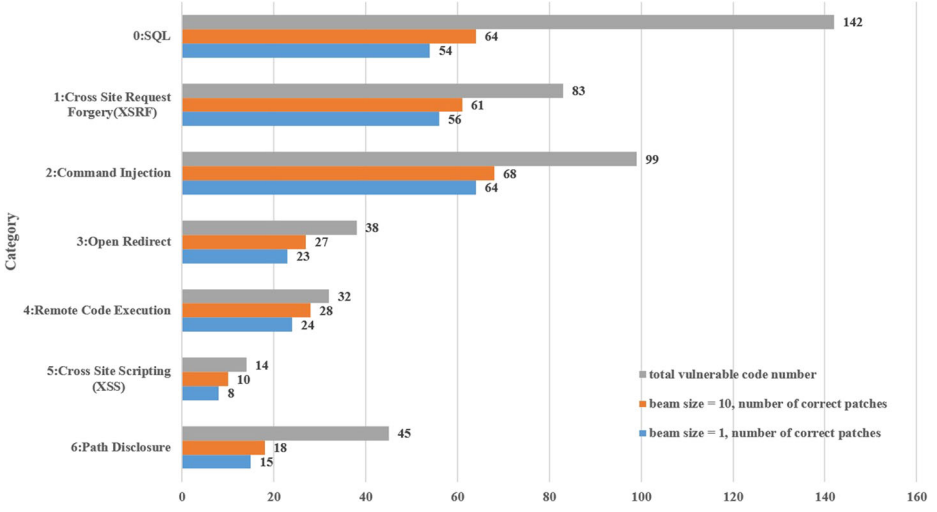


Fig. 8 The number of *correct patch* over categories(Python), value on the horizontal axis corresponds to the category defined in Table 2

```
CWE-476, overflow pointer dereference stream null cdef allow
```

```
static int jp2_cdef_getdata(jp2_box_t *box, jas_stream_t *in)
{
  jp2_cdef_t *cdef = &box->data.cdef;
  jp2_cdefchan_t *chan;
  unsigned int channo;
+ cdef->ents = 0; // SPVF predicts <unk> = <unk>
  ...
  return 0;
}
```

Fig. 9 Case: wrong prediction for improper check or handling of exceptional conditions

Result 3: SPVF excels at fixing Improper Access Control and Incorrect Calculation in C/C++ dataset, Command injection and Cross Site Request Forgery(XSRF) in Python dataset.

7 Threats to Validity

Internal Threats A threat to internal validity lies in the potential faults in implementing our approach. To alleviate the threat, our model is based on the fairseq (Ott et al. 2019) framework to avoid faults in re-implementing. Also, hyperparameter has a huge impact on the final results. To alleviate this threat, we try several sets of hyperparameters besides the hyperparameter recommended by the original NMT model and use the hyperparameters that is capable of generating the best result. Another internal threat lies in the re-implementation of SeqTrans. As sequence-to-sequence model does not need adaptation besides changing the dataset, the potential threat mainly lies in the different tools for constructing the Data Flow Graph.

External Threats In our experiment, our model was evaluated on two public datasets. The datasets regard the committed patch as the correct patch. Therefore, the quality of the public datasets is an external threat to validity as it is unpredictable whether this patch does not introduce new vulnerabilities or whether this patch fails to patch another possible vulnerability. Moreover, in practice, it is uncertain whether SPVF can achieve the same performance when applying vulnerability localization first to get the vulnerable functions before SPVF because it depends on the effectiveness of vulnerability localization. In addition, the vulnerability description might be low quality as in Exploit Database sometimes only vulnerability

```
CWE-119, allow, bound CreateFileResult
```

```
static PassRefPtrWillBeRawPtr<CreateFileResult> create(){
-     return new CreateFileResult();
+     return adoptRefWillBeNoop(new CreateFileResult());
}
AST: FunctionDef, ReturnStatement, ReturnType, ParameterList...
```

Fig. 10 Case: improper control of a resource through its lifetime

```
SQL, session, login
```

```
- resp = requests.get('http://127.0.0.1:5000/profiles')
+ resp = requests.get('http://127.0.0.1:5000/profile/all')""]
AST: Module, Assign,... , Call..
```

Fig. 11 Case:SQL

title and type are provided, thus lowering the performance of SPVF. We then look into the cases that security property is rather short and mainly contains the category information. It turns out in the limited datasets we use, the cases with "low quality" descriptions are not patched significantly worse and some even better. As in Fig. 5, SPVF patches this vulnerable snippet successfully with merely two words "command injection" and "ssh".

8 Related Work

There are many works on APR based on machine learning techniques (Chen et al. 2021; Jiang et al. 2021; Lutellier et al. 2020). The survey (Monperrus 2018) gives a comprehensive overview of program repair techniques. SequenceR (Chen et al. 2021) is a technique for automatic program repair based on sequence to sequence learning. SequenceR considers the context of the bugs and uses an encoder/decoder architecture with the copy mechanism to overcome the unlimited vocabulary problem. CoCoNuT (Lutellier et al. 2020) fixes one-line bugs using the convolutional neural networks (CNN) along with ensemble learning. In the architecture CoCoNuT proposed, the input embedding has a different length from the output embedding because of its architecture consisting of one input encoder, one context encoder along with one output encoder. Our setup is different from these approaches in that the vulnerable function is already analyzed and has its corresponding descriptions including the vulnerability category. Our approach is different from these approaches because we introduce pointer generator for better integration of the security properties. Also, SPVF performs the repair at the function level which is different from the line-level patch for repair.

Several researches for program repair use textual information like compiler error message and build error message. Yasunaga and Liang (2020) proposed an approach for repairing programs from diagnostic feedback. They try to connect symbols relevant to program repair in source code and diagnostic feedback with graph neural network applied to reasoning and then present a self-supervised learning paradigm for program repair. The difference between their work and ours is the way of incorporating the information into the model. Mesbah et al. (2019) proposed a system called DeepDelta that uses diagnostic information for training the Neural Machine Translation Network. Tarlow et al. (2020) proposed Graph2Tocopo which represents the source code, build configuration and compiler diagnostic messages as a graph and predict the diff by feeding the graph into the Graph Neural Network. Abhinav et al. (2021) proposed the RepairNet that uses both code and error messages to repair the program.

For vulnerability fixing, Senx (Huang et al. 2019) uses a set of security properties. They detect the security property violated by the vulnerability input and generates a corresponding patch. Senx only deals with specific class of vulnerability namely integer overflow, buffer overflow and bad cast. The definition for security property in Senx is different from ours in that they refer security property as the techniques relevant to loop cloning and access

range analysis in the process of dealing with vulnerabilities. In addition, SPVF is based on the neural machine learning approaches instead of the designed rules which are used by the Senx.

Harer et al. (2018) investigated the feasibility of using generative adversarial networks (GAN) to fix the software vulnerabilities. Unlike Transformer-based architectures, GAN is not originally built for the task of translation. Yet the authors proposed a loss function especially for NMT tasks in order to tailor the GAN more suitable for the task. Chen et al. (2022) proposed an approach for repairing security vulnerabilities named VRepair which is based on transfer learning. Instead of using the vulnerabilities as training dataset, VRepair is trained on a large bug fix corpus and tuned on a vulnerability fix dataset.

SeqTrans is a tool for automatic vulnerability fixing via sequence to sequence learning (Chi et al. 2020). They proposed to leverage data flow dependencies to construct code sequences and fed them into the Transformer model. The work by Chen et al. (2019) also fixes C vulnerabilities using sequence to sequence learning. Both of the works are based on Transformer. The main difference between their work and ours is that we consider security properties.

9 Conclusion and Future Work

In this paper, we propose a novel approach for automatic vulnerability fixing, called SPVF. SPVF is based on the attention mechanism and extends the attention mechanism with security properties. SPVF makes use of the security properties extracted from the CWE category as well as the natural language description, integrating them with the pointer generator. We evaluated our approach on two datasets. The experimental results show that SPVF outperforms state-of-the-art approaches by 13% for C/C++ and 47% for Python. The further evaluation shows that security property plays a key role in assisting vulnerability fixing and the pointer generator is capable of integrating and utilizing the security properties.

Future Work Our approach uses the "exact same" criterion for evaluation which is not suitable in some cases. We find the case that SPVF fails for not being exactly the same as the target code while it is capable of fixing the vulnerability in a correct way. We would like to explore a more fair way of evaluating approaches to fix the vulnerabilities. Also, we would like to explore more ways of representing code, like the code property graph (CPG) (Yamaguchi et al. 2014).

Acknowledgements This work is supported by the National Natural Science Foundation of China (No.61872312, No.61972335, No.62002309); the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053), the Jiangsu "333" Project; the Natural Science Foundation of the Jiangsu Higher Education Institutions of China (No. 20KJB520016); the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No.KFKT2022B17, No.KFKT2020B16), the Yangzhou University Interdisciplinary Research Foundation for Animal Husbandry Discipline of Targeted Support (yzuxk202015), the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project (YZ2021157) and Yangzhou University Top-level Talents Support Program (2019).

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

Abhinav K, Sharvani V, Dubey A, D'Souza M, Bhardwaj N, Jain S, Arora V (2021) Repairnet: contextual sequence-to-sequence network for automated program repair. In: Roll I, McNamara DS, Sosnovsky SA, Luckin R, Dimitrova V (eds) Artificial intelligence in education - 22nd international conference, AIED 2021, Utrecht, The Netherlands, June 14–18, 2021, Proceedings, Part I, Lecture notes in computer science, vol 12748. Springer, pp 3–15. https://doi.org/10.1007/978-3-030-78292-4_1

Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: 3rd international conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings

Bird S, Loper E (2004) NLTK: the natural language toolkit. In: Proceedings of the 42nd annual meeting of the association for computational linguistics, Barcelona, Spain, July 21–26, 2004 - Poster and Demonstration. ACL

Cao S, Sun X, Bo L, Wei Y, Li B (2021) BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection. Inf Softw Technol 136(106):576. https://doi.org/10.1016/j.infsof.2021.106576

Cao S, Sun X, Bo L, Wu R, Li B, Tao C (2022) MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: 44th IEEE/ACM 44th international conference on software engineering, ICSE 2022, Pittsburgh, PA, USA, May 25–27, 2022. IEEE, pp 1456–1468. https://doi.org/10.1145/3510003.3510219

Chakraborty S, Krishna R, Ding Y, Ray B (2020) Deep learning based vulnerability detection: are we there yet? arxiv:2009.07235

Chen Z, Kommrusch S, Monperrus M (2019) Using sequence-to-sequence learning for repairing C vulnerabilities. arxiv:1912.02015

Chen Z, Kommrusch S, Monperrus M (2022) Neural transfer learning for repairing security vulnerabilities in c code. IEEE Trans Softw Eng. https://doi.org/10.1109/TSE.2019.2940179. arXiv:2104.08308

Chen Z, Kommrusch S, Tufano M, Pouchet L, Poshyvanyk D, Monperrus M (2021) Sequencer: sequence-to-sequence learning for end-to-end program repair. IEEE Trans Software Eng 47(9):1943–1959. https://doi.org/10.1109/TSE.2019.2940179

Cheng X, Wang H, Hua J, Xu G, Sui Y (2021) Deepwukong: statically detecting software vulnerabilities using deep graph neural network. ACM Trans Softw Eng Methodol 30(3):38:1–38:33

Chi J, Qu Y, Liu T, Zheng Q, Yin H (2020) Seqtrans: automatic vulnerability fix via sequence to sequence learning. arxiv:2010.10805. Accessed May 2021

Cooper N, Bernal-Cárdenas C, Chaparro O, Moran K, Poshyvanyk D (2021) It takes two to TANGO: combining visual and textual information for detecting duplicate video-based bug reports. In: 43rd IEEE/ACM international conference on software engineering, ICSE 2021, Madrid, Spain, 22–30 May 2021. IEEE, pp 957–969. https://doi.org/10.1109/ICSE43902.2021.00091

CVE (2021) Common vulnerabilities and exposures. https://cve.mitre.org/. Accessed May 2021

CWE (2021a) Common weakness enumeration. https://cwe.mitre.org/. Accessed May 2021

CWE (2021b) CWE: the category by research concepts. https://cwe.mitre.org/data/definitions/1000.html. Accessed May 2021

Delamore B, Ko RKL (2015) A global, empirical analysis of the shellshock vulnerability in web applications. In: 2015 IEEE Trustcom/bigdataSE/ISPA, Helsinki, Finland, August 20–22, 2015, vol 1. IEEE, pp 1129–1135. https://doi.org/10.1109/Trustcom.2015.493

Durumeric Z, Kasten J, Adrian D, Halderman JA, Bailey M, Li F, Weaver N, Amann J, Beekman J, Payer M, Paxson V (2014) The matter of heartbleed. In: Proceedings of the 2014 internet measurement conference, IMC 2014, Vancouver, BC, Canada, November 5–7, 2014. ACM, pp 475–488. https://doi.org/10.1145/2663716.2663755

Exploit-DB (2021a) Online enrollment management system in php and paypal 1.0. https://www.exploit-db.com/exploits/50557. Accessed Dec 2021

Exploit-DB (2021b) Exploit database. https://www.exploit-db.com/. Accessed Dec 2021

Fan J, Li Y, Wang S, Nguyen TN (2020) A C/C++ code vulnerability dataset with code changes and CVE summaries. In: MSR '20: 17th international conference on mining software repositories, Seoul, Republic of Korea, 29–30 June, 2020. ACM, pp 508–512

Freitag M, Al-Onaizan Y (2017) Beam search strategies for neural machine translation. In: Luong T, Birch A, Neubig G, Finch AM (eds) Proceedings of the first workshop on neural machine translation, NMT@ACL 2017, Vancouver, Canada, August 4, 2017. Association for Computational Linguistics, pp 56–60. https://doi.org/10.18653/v1/w17-3207

Github (2007) Github: a code hosting platform for version control and collaboration. https://github.com/

Harer J, Ozdemir O, Lazovich T, Reale CP, Russell RL, Kim LY, Chin SP (2018) Learning to repair software vulnerabilities with generative adversarial networks. In: Advances in neural information processing systems 31: annual conference on neural information processing systems 2018, NeurIPS 2018, December 3–8, 2018, Montréal, canada, pp 7944–7954

Huang Z, Lie D, Tan G, Jaeger T (2019) Using safety properties to generate vulnerability patches. In: 2019 IEEE symposium on security and privacy, SP 2019, San Francisco, CA, USA, May 19–23, 2019. IEEE, pp 539–554

Jiang N, Lutellier T, Tan L (2021) CURE: code-aware neural machine translation for automatic program repair. In: 43rd IEEE/ACM international conference on software engineering, ICSE 2021, Madrid, Spain, 22–30 May 2021. IEEE, pp 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

Kingma DP, Ba J (2015) Adam: a method for stochastic optimization. In: 3rd international conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings. arXiv:1412.6980

Koyuncu A, Liu K, Bissyandé TF, Kim D, Klein J, Monperrus M, Traon YL (2020) Fixminer: mining relevant fix patterns for automated program repair. Empir Softw Eng 25(3):1980–2024. https://doi.org/10.1007/s10664-019-09780-z

Li B, Wei Y, Sun X, Bo L, Chen D, Tao C (2022) Towards the identification of bug entities and relations in bug reports. Autom Softw Eng 29(1):24. https://doi.org/10.1007/s10515-022-00325-1

Li F, Paxson V (2017) A large-scale empirical study of security patches. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, CCS 2017, Dallas, TX, USA, October 30–November 03, 2017. ACM, pp 2201–2215

Li Y, Wang S, Tien N (2021) Vulnerability detection with fine-grained interpretations, pp 292–303. https://doi.org/10.1145/3468264.3468597

Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: a deep learning-based system for vulnerability detection. arxiv:1801.01681

Lutellier T, Pham HV, Pang L, Li Y, Wei M, Tan L (2020) Coconut: combining context-aware neural translation models using ensemble for program repair. In: ISSTA '20: 29th ACM SIGSOFT international symposium on software testing and analysis, virtual event, USA, July 18–22, 2020. ACM, pp 101–114

Mesbah A, Rice A, Johnston E, Glorioso N, Aftandilian E (2019) Deepdelta: learning to repair compilation errors. In: Dumas M, Pfahl D, Apel S, Russo A (eds) Proceedings of the ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019. ACM, pp 925–936. https://doi.org/10.1145/3338906.3340455

Monperrus M (2018) Automatic software repair: a bibliography. ACM Comput Surv 51(1):17:1–17:24

Ni Z, Li B, Sun X, Chen T, Tang B, Shi X (2020) Analyzing bug fix for automatic bug cause classification. J Syst Softw 163(110):538. https://doi.org/10.1016/j.jss.2020.110538

Ott M, Edunov S, Baevski A, Fan A, Gross S, Ng N, Grangier D, Auli M (2019) Fairseq: a fast, extensible toolkit for sequence modeling. In: Proceedings of the 2019 conference of the North American Chapter of the association for computational linguistics: human language technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2–7, 2019, Demonstrations. Association for Computational Linguistics, pp 48–53

Palo Alto Networks (2021) The state of exploit development: 80% of exploits publish faster than cves. https://unit42.paloaltonetworks.com/state-of-exploit-development/. Accessed Dec 2021

Pradel M, Murali V, Qian R, Machalica M, Meijer E, Chandra S (2020) Scaffle: bug localization on millions of files. In: Khurshid S, Pasareanu CS (eds) ISSTA '20: 29th ACM SIGSOFT international symposium on software testing and analysis, virtual event, USA, July 18–22, 2020. ACM, pp 225–236. https://doi.org/10.1145/3395363.3397356

Python (2021) Python program analysis by Microsoft. https://github.com/microsoft/python-program-analysis

See A, Liu PJ, Manning CD (2017) Get to the point: summarization with pointer generator networks. In: Proceedings of the 55th annual meeting of the association for computational linguistics, ACL 2017, Vancouver, Canada, July 30–August 4, Volume 1: Long Papers. Association for Computational Linguistics, pp 1073–1083

Sun X, Peng X, Zhang K, Liu Y, Cai Y (2019) How security bugs are fixed and what can be improved: an empirical study with mozilla. Sci China Inf Sci 62(1):19,102:1–19,102:3. https://doi.org/10.1007/s11432-017-9459-5

Tarlow D, Moitra S, Rice A, Chen Z, Manzagol P, Sutton C, Aftandilian E (2020) Learning to fix build errors with graph2diff neural networks. In: ICSE '20: 42nd international conference on software engineering, workshops, Seoul, Republic of Korea, 27 June–19 July, 2020. ACM, pp 19–20. https://doi.org/10.1145/3387940.3392181

Transformer (2020) https://en.wikipedia.org/wiki/Transformer_(machine_learning_model). Accessed May 2021

Tufano M, Watson C, Bavota G, Penta MD, White M, Poshyvanyk D (2018) An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018, Montpellier, France, September 3–7, 2018. ACM, pp 832–837. https://doi.org/10.1145/3238147.3240732

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems 30: annual conference on neural information processing systems 2017, December 4–9, 2017, Long Beach, CA, USA, pp 5998–6008

Vudenc (2021) Vulnerability detection with deep learning on a natural codebase. https://github.com/LauraWartschinski/VulnerabilityDetection. Accessed May 2021

Wartschinski L (2019) Vudenc - python corpus for word2vec. https://doi.org/10.5281/ZENODO.3559480. Accessed May 2021

Wei Y, Sun X, Bo L, Cao S, Xia X, Li B (2021) A comprehensive study on security bug characteristics. J Softw Evol Process 33(10). https://doi.org/10.1002/smr.2376

Wikipedia (2021) Scripting language by wikipedia. https://en.wikipedia.org/wiki/Scripting_language. Accessed Dec 2021

Xin Q, Reiss SP (2017) Leveraging syntax-related code for automated program repair. In: Rosu G, Penta MD, Nguyen TN (eds) Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017, Urbana, IL, USA, October 30–November 03, 2017. IEEE Computer Society, pp 660–670. https://doi.org/10.1109/ASE.2017.8115676

Yamaguchi F Joern: a platform for robust analysis of c/c++ code. https://github.com/octopus-platform/joern/tree/master

Yamaguchi F, Golde N, Arp D, Rieck K (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE symposium on security and privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society, pp 590–604. https://doi.org/10.1109/SP.2014.44

Yang S, Wang Y, Chu X (2020) A survey of deep learning techniques for neural machine translation. arxiv:2002.07526

Yasunaga M, Liang P (2020) Graph-based, self-supervised program repair from diagnostic feedback. In: Proceedings of the 37th international conference on machine learning, ICML 2020, 13–18 July 2020, Virtual Event, Proceedings of machine learning research, vol 119, pp 10,799–10,808. PMLR

Zhou C, Li B, Sun X, Bo L (2021) Why and what happened? Aiding bug comprehension with automated category and causal link identification. Empir Softw Eng 26(6):118. https://doi.org/10.1007/s10664-021-10010-8

Zhou Zhou is working towards the BE degree in the School of Information Engineering, Yangzhou University, China. Her research interests include vulnerability fixing based on deep learning.

Lili Bo is currently an associate professor in School of Information Engineering, Yangzhou University, China. She received the PhD degree in School of Computer Science and Technology, China University of Mining and Technology in 2019. Her research interests include software testing, software security.

Xiaoxue Wu is currently a lecturer in School of Information Engineering, Yangzhou University, China. She received the PhD degree in Northwestern Polytechnical University in 2021. Her research interests include software testing, software security.

Xiaobing Sun is currently a professor in School of Information Engineering, Yangzhou University, China. He received the PhD degree in School of computer science and engineering, Southeast University in 2012. His research interests include intelligent software engineering and software data analytics.

Tao Zhang is currently an associate professor in School of Computer Science and Engineering, Macau University of Science and Technology (MUST), Macao SAR, China. He received the BS degree in automation, the M.Eng. degree in software engineering from Northeastern University, China, and the Ph.D. degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. His research interests include AI for software engineering and mobile software security.

Bin Li is a professor in School of Information Engineering, Yangzhou University, China. His current research interests include software engineering, artificial intelligence.

Jiale Zhang is currently a lecture with the School of Information Engineering, Yangzhou University, China. He received the Ph.D. degree in computer science and technology the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2021. His research interests are mainly federated learning, blockchian security, and privacy preserving.

Sicong Cao is currently a PhD student with the School of Information Engineering, Yangzhou University, China. His current research interests include software security and deep learning.

Affiliations

Zhou Zhou[1] · Lili Bo[1,2] · Xiaoxue Wu[1] · Xiaobing Sun[1,2] · Tao Zhang[3] · Bin Li[1] · Jiale Zhang[1] · Sicong Cao[1]

Zhou Zhou
181303130@yzu.edu.cn

Xiaoxue Wu
xiaoxuewu@yzu.edu.cn

Tao Zhang
tazhang@must.edu.mo

Bin Li
lb@yzu.edu.cn

Jiale Zhang
zhangjle@163.com

Sicong Cao
mx120190439@yzu.edu.cn

1 School of Information Engineering, Yangzhou University, Yangzhou, China

2 State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

3 Faculty of Information Technology, Macau University of Science and Technology, Macau, China