EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

VulRep: vulnerability repair based on inducing commits and fixing commits

Ying Wei[1], Lili Bo[1,2*], Xiaoxue Wu[1], Yue Li[1], Zhenlei Ye[1], Xiaobing Sun[1,3] and Bin Li[1]

*Correspondence:
lilibo@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] Jiangsu Province Engineering Research Center of Knowledge Management and Intelligent Service, Yangzhou, China

Abstract

With the rapid development of the information age, software vulnerabilities have threatened the safety of communication and mobile network, and research on vulnerability repair is urgent. Different from the existing machine learning-based approaches, we propose *VulRep*, a vulnerability repair approach based on vulnerability introduction, which combines empirical research findings on vulnerability inducing and vulnerability fixing commit with machine learning approaches for vulnerability repair. Firstly, we construct the vulnerability introduction and repair dataset, and generate the AST tree for the code of inducing commit and fixing commit to form a sequence after abstraction processing, and input it into the *Transformer* model to generate a recommendation list through *beam search*. After filling in the abstracted code, it is combined with the rules defined by empirical research findings, and the final patch is obtained after verification. Experimental results show that *VulRep* can improve the performance of repairing vulnerabilities, which illustrates the effectiveness of combined empirical research findings. In addition, we found that our approach is more suitable for repairing type CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) vulnerabilities and can perform vulnerability repair better.

Keywords: Vulnerability fixing, Software vulnerability, Patch recommendation

1 Introduction

With the development of the information age, software vulnerabilities have become one of the major threats to software security, which exist covertly at all stages of the software life cycle [1–4]. According to statistics, the number of vulnerability reports in the public security database CVE[1] has skyrocketed to nearly 190,000 [5]. This has led to an increasing workload for software developers to manually fix vulnerabilities, which is tedious, time-consuming and error-prone [6–10]. Therefore, many existing researches have explored and analyzed vulnerabilities [11–15], where automatic vulnerability repair remains an important topic, which is of great importance for software maintenance and development.

Automatic vulnerability repair (AVR) is an approach of automatically generating fixing codes to resolve software vulnerabilities, without human intervention [16–18].

[1] https://cve.mitre.org.

Currently, mainstream AVR approaches usually locate the erroneous elements (files, methods, statements) in the vulnerable program firstly, then generate and rank the repair patches using patch generation techniques, and finally recommend the verified correct patches to the developers [19]. Existing AVR techniques are mainly based on heuristic search, statistical analysis, manual fixing templates, and semantic constraints [20]. AVR approaches based on heuristic search [17, 21–23] are the most widely researched class of approaches by far, and it has strong generality. However, both this approach and the statistical analysis-based AVR approach [24–27] require collecting a large number of patches in open-source projects and learning their repair patterns as a way to design repair strategies. As a result, such approaches have huge requirements for patch search space, and this only works well if similar vulnerability fixing code exists in the search space. This makes the search less efficient and increases the time cost of remediation. In addition, AVR approaches based on manual fixing templates [28, 29] and semantic constraints [30, 31] guide patch generation based on manually defined templates or correct statutes of the program by analyzing the patch code. However, in the actual fix process, the code repair or change operation for a class of vulnerabilities is not limited to one fixing template. Therefore, the above two approaches lack flexibility. Moreover, the research on vulnerability fixing templates or protocols is often limited by the available data and lacks comprehensive investigation, making it difficult to perform well when applied to vulnerability fixing.

For the above problems, we propose a vulnerability fixing approach, *VulRep*, based on vulnerability inducing commits and fixing commits. Vulnerability inducing means that when a developer fixes the first vulnerability, the submitted code induces the second vulnerability. Moreover, a commit that inducing a new vulnerability while fixing the previous vulnerability is called a vulnerability inducing commit (*vul-inducing commit*), while a commit that fixes the second vulnerability is called a vulnerability fixing commit (*vul-fixing commit*). In the process of fixing existing vulnerabilities, new vulnerabilities are inevitably induced. We collected 453 vulnerabilities from the CVE database, and all of these data induced another vulnerability in the process of fixing vulnerabilities. In addition, existing research [32] shows that most newly induced vulnerabilities have aggravated in severity, indicating that vulnerability inducing exists and cannot be ignored. However, clarifying the reason for the inducing of the vulnerability can be effective in avoiding the creation of the vulnerability [33, 34], thus pointing the way of a fix. Therefore, *VulRep* takes the vul-inducing commit as a starting point to analyze the fix composition in the vul-inducing commit and the vul-fixing commit. Fix compositions are variable names, method names, and value codes that appear in a normalized and abstracted form when the change operation of the statement to be fixed is known. The proposed approach learns the process of vulnerability fixing so as to generate a fixing template for the vulnerability, after which the relationship between the inducing and the fixing is used to supplement the fix compositions (code) and finally complete the vulnerability fixing. This AVR approach based on vulnerability inducing not only combines the machine learning approach, but also incorporates the findings from empirical studies to compensate for the deficiencies of both. The experimental results show that our approach can effectively repair the induced vulnerabilities. The main contributions of this paper are as follows:

- We combine empirical research findings for vul-inducing commits with machine learning methods, to propose a new vulnerability repair approach based on vulnerability introduction, *VulRep*.
- We explored the relationship between vulnerability introduction and fixing and obtained two findings, to better utilize the code in vul-fixing commits for vulnerability repair.
- We collected vulnerabilities introduced due to commits in CVE as the test dataset, containing a total of 116 pairs of vul-inducing and vul-fixing commits. Experiments based on this dataset demonstrated that the proposed approach combining empirical analysis of vulnerability data is effective.

2 Preliminaries and motivation

2.1 Preliminaries on models

In our proposed vulnerability repair approach, we use the following models and algorithms:

Seq2seq model [35–38] consists of three parts: an encoder, a decoder, and a fixed-size intermediate vector representation that connects the two. In the encoder (e.g., BiLSTM [39]), the code sequence is converted into a vector through the embedding layer and input into an RNN structure, and the overall representation vector is obtained after calculation. In the decoder, each step is to predict the output word of the next state based on the current correct output word and the state of the previous step. In bug repair, DeepFix [24] proposes an end-to-end solution to grammatical bugs in some natural languages, which can fix multiple bugs in one program without relying on any external tools. SequenceR [25] uses Seq2seq for bug repair that address vocabulary limitations in the code. Therefore, the proposed *VulRep* approach uses an end-to-end technique SequenceR to perform vulnerability repair with the AST of the method involved in the vul-fixing commit as input.

Transformer [40] is an attention-based network architecture that completely abandons recursion and convolution. *Transformer* consists of encoder and decoder. Both encoder and decoder contain six blocks. Also take the translation model as an example. When performing a translation task, first obtain the representation vector W of each word in the input sentence (the embedding of the representation vector of the word is added to the embedding of the word position), and input the representation vector matrix of the word into the code. In the encoder, the encoding information of all words in a sentence is obtained after passing through six encoder blocks. In bug fixing, TranS3 [41] encodes collected code snippets based on *Transformer* and *Transformer*'s encoder and decodes a given code snippet to generate its annotations. Tree-LSTM [42] devised a generalized tree-dependent attention framework. What is worth noting in *Transformer* is the attention mechanism it uses. It can effectively alleviate the problem of limited computing power of the model. When a bunch of information carriers appear, how to pay more attention to useful information, based on which an attention mechanism is proposed. Therefore, we chose *Transformer* as the method of learning vulnerability code repair.

Beam search [43, 44] is an algorithm to find the optimal solution in a relatively limited search space with less cost. Suppose the input sequence is X, the

Table 1 Examples of tag lengths for variables and methods

Length of tag	C/C++ language	Java language
1	Put()	Put()
2	mutex_lock()	getName()
3	rcu_read_unlock()	getNextBlock()

output sequence is $Y = (y_1, y_2, \ldots, y_n)$, the probability distribution of decoding is: $P(Y \mid X) = P(y_1 \mid X)P(y_1 \mid y_2, X) \ldots P(y_m|y_1 \ldots m-1|X)$. When decoding, if all possible options are listed, it will lead to an explosion of the space, greedy search is to choose the order of the highest probability each time output. *Beam search* is a compromise method between exhaustive method and greedy search, that is, only the first k possible results are retained in each step of decoding.

2.2 Motivation

Bo et al. [32] study the specific correlation between vul-inducing and vul-fixing commits. Research has shown that fixing commits that induce a new vulnerability are usually caused by incorrect fixes, incomplete fixes, or a combination of both. In order to perform more accurate code filling and more efficient code search, we continue to explore the relationship between vul-inducing and vul-fixing commits on our dataset based on the research results of Bo et al. [32] and obtain the following three observations.

Observation 1 We collected a total of 116 pairs of vul-inducing and vul-fixing commits, and the specific collection process is introduced in Sect. 4.1. Based on this dataset, we conduct the statistical analysis on the method names and variable names in the statements involved in the vul-inducing commits. It can be found that 74.1% of the method names and variable names from vul-inducing commits could be matched in the modified statement of the vul-fixing commits. In the process of learning about vulnerability code fixes, the corresponding fix components need to be generated, so we further explored the code in the vul-fixing commits for better exploitation. We split the method names and variable names in the vul-inducing commits, using different splitting principles (Camelcase and PASCAL nomenclature) as the dataset is multilingual and contains both C/C++ and java languages. The C/C++ language is split using "_" or by symbols. The Java language, on the other hand, can be split by words, for example, the getNextBlock() method can be seen as a variable name consisting of three words, and constants in java are split by upper case letters. To unify the results of the exploration, as shown in Table 1, the tag length of each variable is regarded as divided into several parts, the example shows the name with length 1-3, and so on.

Observation 2 We also counted the percentage of fix components at different lengths as a percentage of fix components in vul-fixing commits. The results show that 50.0% of the fix components tagged with length 1 can be inferred from the introduction of vulnerability in the commit, while 31.9% and 20.7% of the fix components with lengths 2 and 3, respectively. For example, for CVE-2022-41918,[2] the description statement in its

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-41918.

a

```
7426                     */
7427                    release_sock(sk);
7428                    current_timeo =
        schedule_timeout(current_timeo);
7429  +                 if (sk != asoc->base.sk)
7430  +                         goto do_error;
7431                    lock_sock(sk);
7432
7433                    *timeo_p = current_timeo;
```

b

```
7599                     */
7600                    release_sock(sk);
7601                    current_timeo =
        schedule_timeout(current_timeo);
7602  -                 if (sk != asoc->base.sk)
7603  -                         goto do_error;
7604                    lock_sock(sk);
7605
7606                    *timeo_p = current_timeo;
```

Fig. 1 Vul-inducing and Vul-fixing commits of CVE-2017-6353

vul-fixing commit contains "Data Streams," which corresponds to the repair component "DATA_STREAM" of its code. This leads to the second observation that there are fix components that can be inferred from vul-inducing commits at different mark lengths. Therefore, *VulRep* generates vulnerability patches by making code recommendations based on the different vul-inducing commits when filling in the abstract fix components.

Observation 3 We counted the changes in operations between the vulnerability introduction commit and the fixing commit, including the coverage of the same operation and the opposite operation. This means whether the modified statement in the introduction commit is the opposite of or the same as the modified statement in the fixing commit, e.g., the opposite of an insert statement is a delete, and the opposite of an update operation is no change. After the comparison, we found that the coverage of the opposite operation was about 59.5%, and the same operation was 35.3%. Thus, the third finding was obtained that some vulnerability fixes can be inferred from vulnerability introduction, and some vulnerabilities can be fixed by restoring the operations in the vul-inducing commit. Taking CVE-2017-6353[3] as an example, this vulnerability exists because of an incorrect fix for CVE-2017-5986.[4] as shown in Fig. 1, (a) from the vul-inducing commit and (b) is the change in the vul-fixing commit, the *if* statement body added in the vul-inducing commit is removed in the vul-fixing commit, thus achieving the purpose of repairing the vulnerability, visually demonstrating the effectiveness of the finding.

From the above three observations, it can be seen that when the vulnerability is repaired, most of the fix components in the vul-fixing commit can be obtained from the vul-inducing commit, and the opposite operation in the commit is helpful to the

[3] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6353.

[4] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5986.

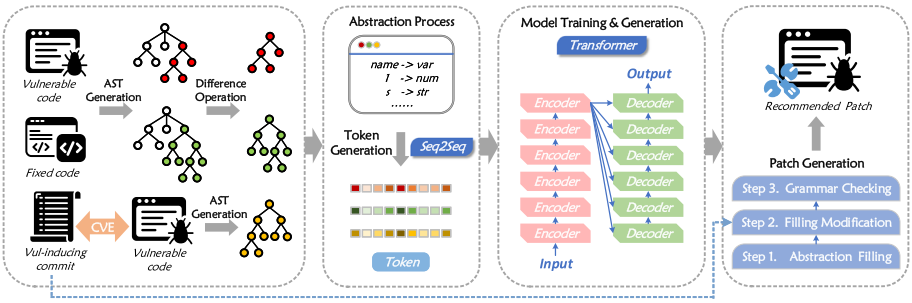

Fig. 2 Overview of our approach *VulRep*

vulnerability repair. Based on the above observations, we propose *VulRep*, a vulnerability repair approach based on vulnerability introduction.

3 Methods

In this paper, we propose *VulRep*, a vulnerability fixing technique based on vul-inducing and vul-fixing commits, the overall framework of which is shown in Fig. 2. Firstly, a vulnerability inducing and fixing pairs dataset is constructed, a syntax tree AST is generated from the code in the inducing and fixing commits, and a difference operation is performed on them. Next, the fix components in the new syntax tree are abstracted and the Seq2Seq model is trained to obtain a token sequence. Then, the token sequences from the training set are fed into the *Transformer* model for training, the test set is converted into token sequences and fed into the trained *Transformer* model, and the list of predicted results is obtained by *beam search*. Finally, the abstract variable and method names are filled in, modified using the rules found through observations, and syntax checked to obtain the final recommended patches. We describe the approach in more details below.

3.1 Vulnerability code tree generation

This section describes the process of generating a vulnerability tree. After constructing the vulnerability inducing and fixing dataset, we processed the code before and after the vulnerability fixes were submitted in the dataset. Firstly, annotations and blank lines were removed from the code files, and the methods involved in the faulty code statements were generated as abstract syntax trees AST's. Secondly, the differences between the two were manipulated, i.e., the differences were used to obtain syntax trees AST's that were closely related to the vulnerabilities. It is important to note that this section chooses to represent the code as an abstract syntax tree, rather than representing the semantics by adding data flow and control flow statements, as is currently common. On the one hand, AST is the most common abstract representation of source code syntactic structures, and it is also more useful for obtaining differences between two pieces of code. On the other hand, Bo et al. [32] explored vulnerability introduction commits and fixing commits and calculated the coverage between the data flow and control flow of a vulnerability introduction commit and the modified statements in a fixing commit. Their results showed that the coverage was low and of little relevance to the vulnerability

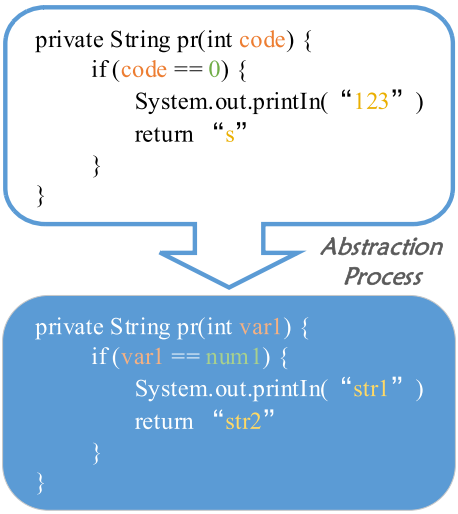

Fig. 3 Example of abstraction process

remediation effort. Therefore, in this approach, we only use abstract syntax trees to represent the contextual structure of the code.

3.2 Code abstraction and serialization

After representing the vulnerability code as an abstract syntax tree, *VulRep* will abstract the code tree due to the fact that when learning the repair of the vulnerability code later, the *Transformer* model takes more into account the probability distribution of a certain word, which forms the vocabulary, in its output. However, when the vulnerable code is put into the model for training, it is prone to vocabulary explosion problems due to the different and large number of variable and method names in the code, naming rules and limitations of the programming language. Therefore, this section abstracts the AST in order to reduce the vocabulary space. As shown in Fig. 3, when variable names, method names and values appear in the code, they are marked abstractly in order, respectively, as *var 1...var n*, *fun 1....fun n* and *num 1... num n*. We then use the Seq2Seq model to learn the abstracted syntax tree and serialize the token sequences required to characterize the *Transformer* model.

3.3 Prediction model training

After abstraction and serialization of the vulnerability code tree, this method feeds Token sequences into the *Transformer* model to learn code repair and use its attention mechanism to alleviate the long dependency problem [26, 40]. OpenAI[5] and DeepMind[6] have used it extensively in their language models [26]. Unlike recurrent neural network (RNN) [45] or long short-term memory (LSTM) [46] models, *Transformer* relies entirely on attention mechanisms to map the global dependencies between input and output data, and thus the model has better translation results. *Transformer* consists of two

[5] https://openai.com/.

[6] https://www.deepmind.com/.

main components: an encoder and a decoder that are connected in series. The structure of the encoder–decoder is widely used in the NMT model, where the encoder maps a sequence of symbolic representations of the input $(x_1, \ldots, x_n)$ to an embedded representation $z = (z_1, \ldots, z_n)$, which contains information about the interrelated parts of the input. Given z, the decoder then uses the merged contextual information to generate the output sequence. At each step, the model uses the previously generated symbols as additional input when generating the next sequence. The *Transformer* follows this overall architecture, using stacked self-attentive layers and point-by-point fully connected layers for the encoders and decoders. Each encoder and decoder uses an attention mechanism to weigh the connections among each input and refer to that information to generate the output. The purpose of the attention mechanism used is to merge the context into the sequence using a set of encodings. The multi-headed attention used in the *Transformer* implements multiple attention mechanisms in parallel and then merges the resultant encodings into a single process. After generating a list of predicted results, we use the *beam search* algorithm to select the appropriate one. The hyperparameter beam width *(B)* in *beam search* indicates the top k results of the ranked sequence are selected. The hyperparameter B in *beam search* is set to 3 in this method in order to maximize performance and save cost.

3.4 Fixed patch generation

The output of *Transformer* differs from the actual compilable code because of the abstraction carried out when processing the data. In this section, the predicted list is abstracted and populated to form the patch after the code has been populated in the order of the dictionary correspondence tokens. However, it is clear from the observations in Sect. 2 that not all vocabularies can be fully represented for the code required in the patch. A vocabulary that is too large tends to result in an explosion of vocabulary space, and conversely, a vocabulary that is too small results in more missing codes and therefore makes it difficult to add the correct repair components. Inspired by the observations, we summarize the specific rules and apply them to the modification of the filled abstract code as follows.

> ***RULE:*** *When the statement in the generated patch is the opposite of or the same as the statement in the vul-inducing commit, update the variable and method names in the patch to the corresponding ones in the vul-inducing commit.*

Using this rule, the generated patch is re-modified to obtain the final vulnerability repair patch. This step is a good combination of the relationship between vul-inducing and vul-fixing commits, which is more conducive to the repair of vulnerabilities. During this process, if the patch generated by the *Transformer* model can pass the syntax check, then this rule will not be used to modify the patch; if it fails, it will be modified according to the rules defined above.

3.5 Fixed patch verification

In this section, the verification of vulnerability recommended patches will be performed manually. Since the vulnerability database CVE does not provide complete vulnerability data, the dataset does not contain test cases related to vulnerabilities. Therefore, we

use the corresponding fixing code in the CVE database as the ground truth and employ the method in [44] to manually check the correctness of the patches generated by VulRep. That is, it checks whether the final patch code provided by the model has semantic equivalence with the vulnerability fixing code submitted in the dataset. In order to better improve the reliability of this step, two graduate students who study this field conduct inspections separately, and the correctness of this patch will be recognized only when the inspections pass. At the same time, we carried out the consistency evaluation according to the evaluation standard of Cohen's Kappa coefficient. For the patch to be verified, Cohen's Kappa coefficient is greater than 80%, that is, the consistency strength is "Almost Perfect." Therefore, it can be considered that there is almost a perfect agreement between the two participants. Until this point, we recommend approved patches to developers to fix vulnerabilities. At the same time, the Cohen's Kappa [47] consistency check was carried out, and the result was 0.95, which is a high consistency. Until this point, we recommend approved patches to developers to fix vulnerabilities.

4 Results and discussion

To verify the validity of the proposed approach *VulRep*, we constructed the corresponding dataset and validated it with two experimental questions. We will describe them in detail below.

4.1 Dataset

In this approach, we used a total of two datasets for training and testing:

(1) *Training dataset* The training data was selected as the *Transformer* model was used to learn the code changes committed by the vulnerability fixes. The training data only requires the code before and after the fix, not the commit description of the vulnerability. Therefore, the publicly available dataset Big-Vul [48] was chosen as our training data, containing a total of 3754 vulnerability data.

(2) *Testing dataset* Bo et al. [32] collected 71 pairs of commits that induced a vulnerability due to an incorrect fix, an incomplete fix, or neither an incorrect nor an incomplete fix. However, our approach is not limited to these types of data, we also focus on vulnerabilities that induced by fixing commits. For example, the description of CVE-2015-6250[7] is: "simple-php-captcha before commit 9d65a945029c7be7bb6bc893759e74c5636be694 allows remote attackers to automatically generate the captcha response by running the same code on the client-side." It indicates that this vulnerability is caused by a commit. The reference links in the vulnerability report provide code commit data in GitHub or other fixing links. We also use some developer comments to determine if a commit is a fix for a vulnerability. Developers who fix vulnerabilities will leave comments on GitHub or other sites such as Red Hat Bugzilla[8] that contain information such as a commit that introduces or fixes a vulnerability, which may be considered a vul-inducing commit. Based on this approach, a total of 45 pairs of vulnerability introductions and fixes were added, resulting in 116 pairs of vul-inducing and vul-fixing commits as the testing dataset for the experiment.

[7] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6250.

[8] https://bugzilla.redhat.com/.

4.2 Research questions

We design experiments to verify the effectiveness of the proposed approach and explore the performance of the approach on different types of vulnerabilities.

RQ1 Compared with the machine learning generated remediation code, how effective is the approach VulRep proposed in this paper?

In most of the existing studies, the generation of vulnerability repair code has been implemented with the help of machine learning approaches. However, our proposed approach, *VulRep*, combines the generation of fixing codes using the *Transformer* with empirical findings to produce the final fixing code. Therefore, we use experiment to investigate whether the findings from empirical research can be useful for vulnerability remediation. The experiments compares *VulRep* with patch generation using machine learning (*Transformer*) alone to see if it yields more effective patches for vulnerability repair.

RQ2 Which type of vulnerabilities does VulRep fix best?

After validating the effectiveness of the proposed approach, we also explored which type of vulnerabilities *VulRep* is better at fixing. On the one hand, some studies will focus on the types of vulnerabilities to fix, such as pointer exceptions, missing specified encoding and other vulnerability types. On the other hand, a statistical study by Bo et al. [32] shows that the logic error type accounts for 66% of the vulnerability data, so vulnerabilities may be relatively clustered in the same type. Therefore, in RQ2, we investigate the correctness of *VulRep*'s fix patches for different vulnerability types. For the types of vulnerabilities, we chose the CWE (Common Weakness Enumeration) standard,[9] which is a classification system for software defects and vulnerabilities [26] and can cover most types of vulnerabilities.

4.3 Evaluation metrics

Since in the process of vulnerability repair, it is necessary to judge whether a generated patch actually fixes the vulnerability or not. The patching situation contains complete fixes, partial fixes and incorrect fixes. Therefore, for patches generated using *VulRep* for vulnerability repair, two metrics are used in this chapter to evaluate the capability and quality of the generated patches, *Precision* and *Recall*, which are calculated as follows:

$$\text{Precision} = \frac{\text{CF}}{\text{CF} + \text{PF}} \tag{1}$$

$$\text{Recall} = \frac{\text{CF}}{\text{CF} + \text{PF} + \text{IF}} \tag{2}$$

where CF denotes the number of completely fixed vulnerabilities, PF denotes the number of partially fixed vulnerabilities, IF denotes the number of incorrectly fixed

[9] https://cwe.mitre.org/.

Table 2 VulRep versus *Transformer* vulnerability repair performance

Approach	Precision	Recall
VulFix	13/23 (56.5%)	13/116 (11.2%)
Transformer	11/20 (55.0%)	11/116 (9.48%)

Table 3 Type and number of vulnerabilities in testing dataset

NO.	CWE-ID	Vul	NO.	CWE-ID	Vul	NO.	CWE-ID	Vul
1	CWE-119	13	7	CWE-369	7	13	CWE-125	6
2	CWE-476	11	8	CWE-79	6	14	CWE-787	5
3	CWE-199	8	9	CWE-200	6	15	CWE-399	5
4	CWE-199	5	10	CWE-189	5	16	CWE-400	4
5	CWE-199	4	11	CWE-416	3	17	CWE-415	3
6	CWE-199	2	12	CWE-120	2	18	CWE-264	3

vulnerabilities, and the sum of the three denotes the total number of vulnerability samples.

4.4 Experiment results

In this section, we observe, analyze and summarize the relevant experimental results for two research questions designed.

RQ1 Effectiveness of VulRep

This paper proposes a new repair approach *VulRep*, that is, learning the difference between the abstract syntax trees of inducing commit and fixing commit, the code in inducing code is used for abstract reduction when generating the patch sequence. Table 2 shows the comparison results of *VulRep* and *Transformer*. The *Precision* of *VulRep* is 56.5%, and the *Precision* of the machine learning approach (*Transformer*) is 55%. The former fixes one more vulnerability than the latter. The *Recall* of *VulRep* and *Transformer* are 11.2% and 9.48%, respectively. It can be concluded that this chapter proposes one more partially repaired vulnerability than the machine learning method. This shows that only using the findings of empirical research can also play a role in repairing some sentences. From a macro-repair perspective, the approach *VulRep* proposed in this paper makes up for the shortcomings of machine learning repair approaches to a certain extent, and also proves that the approach of combining vul-inducing commits with machine learning approaches to repair vulnerabilities is effective.

The comparison of the two approaches proves the validity of the rule defined by the empirical research findings for the vulnerability repair and also verifies the correctness of the empirical research in Sect. 2. Although the current mainstream approaches are all based on machine learning, it turns out that empirical analysis of

Table 4 Types of vulnerabilities fixed by *VulRep*

Top	CWE-ID	Vulnerabilities
Top1	CWE-119	6
Top2	CWE-476	3
Top3	CWE-200	1

vulnerability data is equally important and cannot be ignored, and the rich information contained in vulnerability submissions is also helpful for vulnerability repair.

Summary of RQ1 Compared to Transformer, VulRep, which is based on vulnerability introduction, shows better performance in vulnerability repair.

RQ2 Performance of VulRep on different types

First, we counted the CWE types of vulnerabilities in the test set, and the results are shown in Table 3. It is important to note that a vulnerability may belong to multiple CWE types. In this table, we have omitted the CWE type with a count of 1 and the 11 vulnerabilities with type none.

From the results of the RQ1, it can be seen that a total of 13 of the fixes obtained through *VulRep* are completely correct. We examined the types of these 13 vulnerabilities, and the results are shown in Table 4. The experiments show that *VulRep* performs best in the case of CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), fixing six vulnerabilities of the relevant type, followed by three fixes for vulnerabilities of CWE-476 (NULL Pointer Dereference). There may be two reasons for this. On the one hand, CWE-119 type vulnerabilities usually involve data boundary or data type checks, etc. These code modifications are usually less and relatively easy. On the other hand, the corresponding data that needs to be modified is usually mentioned in the commit, and the code vocabulary involved in this type of vulnerability is also relatively concentrated, such as length and size. For example, the "body size" in the commit can be mapped to the "body_size" in the repair code. The rules used by *VulRep* can use this correspondence to assist in the repair, so it has a better performance on CWE-119 types excellent. Moreover, it also fixed a CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor) of vulnerability. Three of the vulnerabilities fixed were of type *none* and are therefore omitted from the table. By looking at the actual patch generation data for the vulnerabilities, we find that indeed, as observed in the empirical study in Sect. 2, the introduction of the commit statement for the opposite operation is of importance. Moreover, the experimental results not only prove the validity of the empirical findings, but also demonstrate that the relationship between vul-inducing and vul-fixing commits can indeed assist in vulnerability repair effectively.

Summary of RQ2 VulRep is best at fixing vulnerabilities of type CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer).

4.5 Threads to validity

Our findings of experiments are based on the *Precision* and *Recall* metrics. Other evaluation metrics may yield different results, these metrics have been widely used to vulnerability repair task.

Internal threats (1) The empirical data used in this paper is relatively small. Although the effectiveness of the summarized repair rules has been demonstrated through experiments, there are some rules being missing inevitably. (2) The proposed approach first generates a repair patch based on machine learning, but it cannot be sure whether it has learned the correct repair pattern. Our approach does not use predefined patterns, which makes fixing certain types of vulnerabilities less effective. (3) After getting the vulnerability patch, this paper uses manual verification to check the correctness, which may be subject to certain degree. However, we verify the validity of the test by calculating Cohen's Kappa coefficient to reduce the threat.

External threats (1) During the evaluation of the experiments, we only collected 116 pairs of data with the introduction of vulnerabilities as a testing dataset. We will expand the dataset in the future to meet the new needs of developers. (2) We use the dataset *Big-Vul* to learn vulnerability code fixes, the number and types of which are still limited, and it may be difficult to meet the training requirements of the model (e.g., the size and number of types of dataset *Big-Vul*) are not large enough, resulting in the model is difficult to achieve fitting. In the future, we will try to manually expand *Big-Vul* to achieve better fitting.

5 Related work

In recent years, a large number of automatic repair techniques for program defects have been proposed, which can be roughly divided into four categories: automatic repair techniques based on heuristic search, statistical analysis, manual fixing templates, and semantic constraints [20]. Zhang et al. [49] applied the idea of search-based program repair to the field of heterogeneous computing and proposed *HeteroGen*, which takes C/C++ code as input and automatically generates a version of HLS with test behavior retention and better performance. Li et al. [50] designed a novel fault location (FL) technique for multi-block, multi-statement repair that combines traditional spectrum-based (SB) FL with deep learning and data flow analysis. Chi et al. [26] provided a novel approach called SeqTrans to exploit historical vulnerability fixes to provide suggestions and automatically fix the source code. It leveraged data-flow dependencies to construct code sequences and feed them into the state-of-the-art transformer model, to capture the contextual information around the vulnerable code. Liu et al. [51] constructed an APR tool, *TBar*, by surveying the literature to collect, summarize and label frequently used repair patterns, which integrates a rich set of repair templates from previous authors, and experimental results show that it can correctly repair a wider range of bugs. Ke et al. [52] proposed *SearchRepair*, a defect repair technique based on semantic search of code, and built a database of code fragments. The above methods need to collect a large number of patches to learn their repair modes. Chen et al. [27] proposed an approach for repairing security vulnerabilities named *VRepair* which is based on transfer learning. *VRepair* is first trained on a large bug fix corpus and is then tuned on a

vulnerability fix dataset, which is an order of magnitude smaller. This approach alleviates the problem of dataset scarcity to some extent. However, existing approaches guide patch generation by analyzing the patch code and manually defining templates or program rules, but this is often incomplete and lacks generalizability and flexibility. Different from them, *VulRep* combine the vul-inducing commit with the machine learning method through the empirical research on the vulnerability introduction and correct the generated code according to the rules to obtain the final patch. The approach is not limited to a certain type of vulnerability and can be more effectively tailored to the characteristics of the vulnerability itself, making it widely available and more flexible.

6 Conclusion

In this paper, the empirical research findings based on vulnerability introduction are combined with machine learning to perform vulnerability repair, and a vulnerability repair approach based on vulnerability introduction (*VulRep*) is proposed. This approach processes the vulnerability repair submission as a sequence, then inputs it into the *Transformer* model, and generates a recommendation list through *beam search*, abstracts and fills the abstract code in the recommendation list, and combines it with the rules defined by empirical research findings to get the final patch. The experimental results show that *VulRep* can effectively improve the effect of repairing the vulnerability, and it performs best in the inappropriate operation restriction (CWE-119) within the scope of the vulnerability type memory buffer, and completes the repair work better.

In the future, we will further expand the existing dataset to meet follow-up research. Furthermore, we will consider automatically extracting the content of vulnerability text to assist machine learning methods in vulnerability repair.

Abbreviations

AVR	Automatic vulnerability repair
CVE	Common vulnerabilities and exposures
CWE	Common weakness enumeration
Vul-inducing	Vulnerability inducing
Vul-fixing	Vulnerability fixing

Acknowledgements
None.

Author contributions
In this paper, YW conceived, designed and wrote the study. LB, XW, XS and BL provided valuable advice on the methodology of the study and participated in the revision of the manuscript. YL and ZY participated in the data collection and manually checked the correctness of the experimental results. All authors read and approved the final manuscript.

Funding
This paper is supported by the National Natural Science Foundation of China (Nos. 61972335, 61872312, 62002309); the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053), the Jiangsu "333" Project; the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No. KFKT2022B17), the Yangzhou University Interdisciplinary Research Foundation for Animal Husbandry Discipline of Targeted Support (No. yzuxk202015), the Yangzhou city - Yangzhou University Science and Technology Cooperation Fund Project (Nos. YZ2021157, YBK202207) and Yangzhou University Top-level Talents Support Program (2019).

Data availability
The datasets used and/or analyzed during the current study are confidential and not public.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 3 January 2023 Accepted: 1 April 2023
Published online: 21 April 2023

References

1. B. Li, Y. Wei, X. Sun, L. Bo, D. Chen, C. Tao, Towards the identification of bug entities and relations in bug reports. Autom. Softw. Eng. **29**(1), 1–31 (2022)
2. Z. Ni, L. Bo, B. Li, T. Chen, X. Sun, X. Wu, An approach of method-level bug localization. IET Softw. **16**, 422–437 (2022)
3. J. Lu, X. Sun, B. Li, L. Bo, T. Zhang, Beat: considering question types for bug question answering via templates. Knowl. Based Syst. **225**, 107098 (2021)
4. S. Cao, X. Sun, L. Bo, R. Wu, B. Li, C. Tao, Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks. arXiv preprint arXiv:2203.02660 (2022)
5. Y. Wei, X. Sun, L. Bo, S. Cao, X. Xia, B. Li, A comprehensive study on security bug characteristics. J. Softw. Evol. Process **33**(10), 2376 (2021)
6. T. Zhou, X. Sun, X. Xia, B. Li, X. Chen, Improving defect prediction with deep forest. Inf. Softw. Technol. **114**, 204–216 (2019)
7. X. Sun, X. Peng, K. Zhang, Y. Liu, Y. Cai, How security bugs are fixed and what can be improved: an empirical study with Mozilla. Sci. China Inf. Sci. **62**(1), 1–3 (2019)
8. Z. Zhou, L. Bo, X. Wu, X. Sun, T. Zhang, B. Li, J. Zhang, S. Cao, Spvf: security property assisted vulnerability fixing via attention-based models. Empir. Softw. Eng. **27**(7), 1–28 (2022)
9. S. Cao, X. Sun, L. Bo, Y. Wei, B. Li, Bgnn4vd: constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. **136**, 106576 (2021)
10. Y. Yin, Y. Li, H. Gao, T. Liang, Q. Pan, FGC, GCN based federated learning approach for trust industrial service recommendation. IEEE Trans. Ind. Inform. **19**(3), 3240–3250 (2022)
11. H. Gao, W. Huang, T. Liu, Y. Yin, Y. Li, Ppo2: location privacy-oriented task offloading to edge computing using reinforcement learning for intelligent autonomous transport systems. IEEE Trans. Intell. Transp. Syst. 1–14 (2022)
12. Q.-V. Dang, Improving the performance of the intrusion detection systems by the machine learning explainability. Int. J. Web Inf. Syst. **17**(5), 537–555 (2021)
13. H. Gao, J. Huang, Y. Tao, W. Hussain, Y. Huang, The joint method of triple attention and novel loss function for entity relation extraction in small data-driven computational social systems. IEEE Trans. Comput. Soc. Syst. **9**(6), 1725–1735 (2022)
14. A.K.Y.S. Mohamed, D. Auer, D. Hofer, J. Küng, A systematic literature review for authorization and access control: definitions, strategies and models. Int. J. Web Inf. Syst. (ahead-of-print) (2022)
15. X. Ma, H. Xu, H. Gao, M. Bian, W. Hussain, Real-time virtual machine scheduling in industry iot network: a reinforcement learning method. IEEE Trans. Ind. Inf. **19**(2), 2129–2139 (2022)
16. M. Monperrus, Automatic software repair: a bibliography. ACM Comput. Surv. (CSUR) **51**(1), 1–24 (2018)
17. C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2011)
18. X. Sun, T. Zhou, R. Wang, Y. Duan, L. Bo, J. Chang, Experience report: investigating bug fixes in machine learning frameworks/libraries. Front. Comput. Sci. **15**(6), 1–16 (2021)
19. H. Cao, Y. Meng, J. Shi, L. Li, T. Liao, C. Zhao, A survey on automatic bug fixing, in *2020 6th International Symposium on System and Software Reliability (ISSSR)* (IEEE, 2020), pp. 122–131
20. C.J. Jiang JiaJun, X. Yingfei, Survey of automatic program repair techniques. J. Softw. **32**(9), 2665–2690 (2021)
21. S. Forrest, T. Nguyen, W. Weimer, C. Le Goues, A genetic programming approach to automated software repair, in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (2009), pp. 947–954
22. C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer, A systematic study of automated program repair: fixing 55 out of 105 bugs for \$8 each, in *2012 34th International Conference on Software Engineering (ICSE)* (IEEE, 2012), pp. 3–13
23. J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018), pp. 298–309
24. R. Gupta, S. Pal, A. Kanade, S. Shevade, Deepfix: fixing common c language errors by deep learning, in *Thirty-First AAAI Conference on Artificial Intelligence* (2017)
25. Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, M. Monperrus, Sequencer: sequence-to-sequence learning for end-to-end program repair. IEEE Trans. Softw. Eng. **47**(9), 1943–1959 (2019)
26. J. Chi, Y. Qu, T. Liu, Q. Zheng, H. Yin, Seqtrans: automatic vulnerability fix via sequence to sequence learning. IEEE Trans. Softw. Eng. **49**, 554–585 (2022)
27. Z. Chen, S. Kommrusch, M. Monperrus, Neural transfer learning for repairing security vulnerabilities in c code. IEEE Trans. Softw. Eng. **49**(1), 147–165 (2022)
28. D. Kim, J. Nam, J. Song, S. Kim, Automatic patch generation learned from human-written patches, in *2013 35th International Conference on Software Engineering (ICSE)* (IEEE, 2013), pp. 802–811
29. J. Hua, M. Zhang, K. Wang, S. Khurshid, Towards practical program repair with on-demand candidate generation, in *Proceedings of the 40th International Conference on Software Engineering* (2018), pp. 12–23
30. J. Xuan, M. Martinez, F. Demarco, M. Clement, S.L. Marcote, T. Durieux, D. Le Berre, M. Monperrus, Nopol: automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. **43**(1), 34–55 (2016)
31. S. Mechtaev, J. Yi, A. Roychoudhury, Angelix: scalable multiline program patch synthesis via symbolic analysis, in *Proceedings of the 38th International Conference on Software Engineering* (2016), pp. 691–701
32. L. Bo, Y. Li, X. Sun, X. Wu, B. Li, Vulloc: vulnerability localization based on inducing commits and fixing commits. Front. Comput. Sci. **17**(3), 1–3 (2023)
33. Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, X. Shi, Analyzing bug fix for automatic bug cause classification. J. Syst. Softw. **163**, 110538 (2020)

34. C. Zhou, B. Li, X. Sun, L. Bo, Why and what happened? aiding bug comprehension with automated category and causal link identification. Empir. Softw. Eng. **26**(6), 1–36 (2021)
35. S. Karaivanov, V. Raychev, M. Vechev, Phrase-based statistical translation of programming languages, in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (2014), pp. 173–184
36. A.T. Nguyen, T.T. Nguyen, T.N. Nguyen, Divide-and-conquer approach for multi-phase statistical migration for source code (t), in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015). IEEE, pp. 585–596
37. A.T. Nguyen, H.A. Nguyen, T.T. Nguyen, T.N. Nguyen, Statistical learning approach for mining api usage mappings for code migration, in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (2014), pp. 457–468
38. X. Chen, C. Liu, D. Song, Tree-to-tree neural networks for program translation, in *Advances in Neural Information Processing Systems*, vol. 31 (2018)
39. P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, B. Xu, Attention-based bidirectional long short-term memory networks for relation classification, in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7–12, 2016, Berlin, Germany, Volume 2: Short Papers* (The Association for Computer Linguistics, 2016)
40. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in *Advances in Neural Information Processing Systems*, vol. 30 (2017)
41. Wang, W., Zhang, Y., Zeng, Z., Xu, G.: Trans ̂3: a transformer-based framework for unifying code summarization and code search. corr abs/2003.03238 (2020). arXiv preprint arXiv:2003.03238 (2020)
42. M. Ahmed, M.R. Samee, R.E. Mercer, Improving tree-LSTM with tree attention, in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)* (IEEE, 2019), pp. 247–254
43. M. Freitag, Y. Al-Onaizan, Beam search strategies for neural machine translation. arXiv preprint arXiv:1702.01806 (2017)
44. V. Raychev, M. Vechev, E. Yahav, Code completion with statistical language models, in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), pp. 419–428
45. W. Zaremba, I. Sutskever, O. Vinyals, Recurrent neural network regularization. arXiv preprint arXiv:1409.2329 (2014)
46. X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, W.-c. Woo, Convolutional LSTM network: a machine learning approach for precipitation nowcasting, in *Advances in Neural Information Processing Systems*, vol. 28 (2015)
47. J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data. Biometrics **33**, 159–174 (1977)
48. J. Fan, Y. Li, S. Wang, T.N. Nguyen, Ac/c++ code vulnerability dataset with code changes and cve summaries, in *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), pp. 508–512
49. Q. Zhang, J. Wang, G.H. Xu, M. Kim, Heterogen: transpiling c to heterogeneous hls code with automated test generation and program repair, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 1017–1029
50. Y. Li, S. Wang, T.N. Nguyen, Dear: a novel deep learning-based approach for automated program repair. arXiv preprint arXiv:2205.01859 (2022)
51. K. Liu, A. Koyuncu, D. Kim, T.F. Bissyandé, Tbar: revisiting template-based automated program repair, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 31–42
52. Y. Ke, K.T. Stolee, C. Le Goues, Y. Brun, Repairing programs with semantic code search (t), in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (IEEE, 2015), pp. 295–306

Publisher's Note