SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning

Jianlei Chi, Yu Qu, Ting Liu, *Member, IEEE,* Qinghua Zheng, *Member, IEEE,* Heng Yin, *Member, IEEE*

Abstract—Software vulnerabilities are now reported unprecedentedly due to the recent development of automated vulnerability hunting tools. However, fixing vulnerabilities still mainly depends on programmers' manual efforts. Developers need to deeply understand the vulnerability and affect the system's functions as little as possible.
In this paper, with the advancement of Neural Machine Translation (NMT) techniques, we provide a novel approach called SeqTrans to exploit historical vulnerability fixes to provide suggestions and automatically fix the source code. To capture the contextual information around the vulnerable code, we propose to leverage data-flow dependencies to construct code sequences and feed them into the state-of-the-art transformer model. The fine-tuning strategy has been introduced to overcome the small sample size problem. We evaluate SeqTrans on a dataset containing 1,282 commits that fix 624 CVEs in 205 Java projects. Results show that the accuracy of SeqTrans outperforms the latest techniques and achieves 23.3% in statement-level fix and 25.3% in CVE-level fix. In the meantime, we look deep inside the result and observe that the NMT model performs very well in certain kinds of vulnerabilities like CWE-287 (Improper Authentication) and CWE-863 (Incorrect Authorization).

Index Terms—Software engineering, vulnerability fix, neural machine translation, machine learning

1 Introduction

Software evolves quite frequently for numerous reasons such as deprecating old features, adding new features, refactoring, bug fixing, etc. Debugging is one of the most time-consuming and painful processes in the entire software development life cycle (SDLC). A recent study indicates that the debugging component can account for up to 50% of the overall software development overhead, and the majority of the debugging costs come from manually checking and fixing bugs [1]. This leads to a growing number of researchers working on teaching machines to automatically modify and fix the program, which is called automated program repair [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14].

A software vulnerability is one kind of bug that can be exploited by an attacher to cross authorization boundaries in the source code. Vulnerabilities like HeartBleed [15], Spectre [16] and Meltdown [17], introduced significant threats to millions of users. Nevertheless, identifying and fixing vulnerabilities is more challenging than bugs [18], [19], [20]. Firstly, the number of vulnerabilities is fewer than bugs, making learning enough knowledge from historical data more difficult. In other words, we usually have only a relatively small database of vulnerabilities. Secondly, labeling and identifying vulnerability requires a mindset of the attacker that may not be available to developers [21]. Thirdly, vulnerabilities are reported at an unprecedented speed due to the recent development of automated vulnerability hunting tools like AFL [22], AFLGo [23], AFLFast [24]. Nevertheless, fixing vulnerabilities depends heavily on manually generating repair templates and defining repair rules, which are tedious and error-prone [25]. Automatically learning to generate vulnerability fixes is urgently needed and will significantly improve the efficiency of software development and maintenance processes.

There are many works of automated program repair (APR) or called code migration in both industrial and academic domains [5]. Some APR studies focus on automatically generating fix templates or called fix patterns [26], [27], [28], [29], [30]. Some of APR studies focus on mining similar code changes from historical repair records such as CapGen [31] and FixMiner [32]. Other approaches utilize static and dynamic analysis with constraining solving to accomplish patch generation [7], [33]. IDEs also provide specific kinds of automatic changes [34]. For example, refactoring, generating getters and setters, adding override/implement methods or other template codes, etc. Recently, introducing Machine Learning (ML) techniques into program repair has also attracted a lot of interest and became a trend [35], [36], [37], [38], which build generic models to capture statistical characteristics using previous code changes and automatically fixing the code being inserted.

However, although some promising results have been achieved, current studies of automated program repair face a list of limitations, especially on fixing vulnerabilities. Firstly, most APR approaches heavily rely on domain-specific knowledge or predefined change templates, which leads to limited scalability [5]. Tufano's dataset [39] contains 2 million sentence pairs of historical bug fix records. Nevertheless, a vulnerability fix dataset such as Ponta's dataset [40] and the AOSP dataset [41] only contain 624 and 1380 publicly disclosed vulnerabilities. The confirmed

- *J. Chi, T. Liu and Q. Zheng are with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), School of Computer Science and Technology, Xian Jiaotong University, Xian 710049, China.*
  Email: chijianlei@stu.xjtu.edu.cn, tliu, qhzhengg@xjtu.edu.cn.
- *Y. Qu and H. Yin are with the Department of Computer Science and Engineering, UC Riverside, California, USA.*
  Email: yuq@ucr.edu, heng@cs.ucr.edu

CVE records number is nearly 150K [1]. This means we need to train and learn from a small dataset of vulnerabilities. Secondly, traditional techniques leverage search space exploration, statistical analysis to rank similar repair records [42]. These techniques need to define large numbers of features, which can be time-consuming and not accurate enough. ML models can alleviate these problems but as mentioned above, only a few studies have been done to focus on vulnerability fixing because of the small sample size.

In this paper, we focus on the two issues raised above and rely entirely on machine learning to capture grammatical and structural information as common change patterns. In order to solve the small sample size problem, we use the fine-tuning method [43]. Fine-tuning means that if our specialized domain dataset is similar to the general domain dataset, we can take weights of a trained neural network and use it as initialization for a new model being trained on data from the same domain. It has been widely utilized to speed up the training and overcome the small sample size. Using this method, we can combine two related works together: vulnerability fixing and bug repair. We will first pre-train the model based on the large and diverse dataset from bug repair records to capture universal features. Then, we will fine-tune the model on our minor vulnerability fixing dataset, freeze or optimize some of the pre-trained weights to make the model more suitable for vulnerability fixing work.

We choose the general approach of Neural Machine Translation (NMT) to learn rules from historical records and apply them to future edits. It is widely utilized in Natural Language Processing (NLP) domain, such as translating one language (e.g., English) to another language (e.g., Swedish). The NMT model can generalize numerous sequence pairs between two languages, learn the probability distribution of changes, and assign higher weights to appropriate editing operations. Previous studies such as Tufano et al. [37] and Chen et al. [38] have shown an initial success of using the NMT model for predicting code changes.

However, they only focus on simple scenarios such as short sequences and single-line cases. Since the NMT model is originally exploited for natural language processing, there is a distinction between natural language and programming language [44]. Firstly, program language falls under the category of language called context-sensitive languages. Dependencies in one statement may come from the entire function or even the entire class. Nevertheless, in natural language, token dependencies are always distributed in the same or neighboring sentences. Secondly, the vocabulary of natural languages is filled with conceptual terms. The vocabulary of programming languages is generally only grammar words like essential comments, plus various custom-named things like variables and functions. Thirdly, programming languages are unambiguous, while natural languages are often multiplied ambiguous and require interpretation in context to be fully understood.

In order to solve the dependency problem across the entire class, we construct the define-use (def-use) [45] chain, which represents the data-flow dependencies to capture important context around the vulnerable statement. It will extract all variable definitions from the vulnerable statements. We use the state-of-the-art transformer model [46] to reduce the performance degradation caused by long statements. This enables us to process long statements and captures a broader range of dependencies.

We called our approach SeqTrans, and it works as follows: Firstly, we collect historical bug and vulnerability fixing records from two previous open datasets, which contain 2 million and 5k sentence pairs of confirmed fix records. Secondly, we start by training a transformer model with a self-attention mechanism [46] for bug repairing on the big dataset. Then, we fine-tune the model on the small dataset to match the target of our work for vulnerability fixing. Thirdly, if a new vulnerable object is inputted to the trained model, beam search [47] will be utilized first to obtain a list of candidate predictions. Then, a syntax checker will filter the candidate list and select the most suitable prediction.

In order to evaluate our approach, we calculate the accuracy at statement level and across the CVE on Ponta's dataset [40]. The experimental result shows that our approach SeqTrans reaches a promising accuracy of single line prediction by 23.3% when Beam=50, outperforms the state-of-the-art model SequenceR [38] by 5% and substantially surpasses the performance Tufano et al. [37] and other NMT models. As for predicting the full CVE, our approach also achieves the accuracy of 25.3% when Beam=50, which is also better than other approaches. We also conducted a traditional evaluation experiment to verify our actual performance. The result shows that among the 120 CVEs we select from 5 open-source projects, we correctly fix 21 of them. We believe these promising results can confirm that SeqTrans is a competitive approach that achieves good performance on the task of vulnerability fixing.

In the meantime, we also made some ablation studies and observed internally what SeqTrans could well predict types of vulnerability fixes. An interesting observation we find is that our model gives results that vary for different types of CWEs. Our model performs quite well in specific types of CWEs like CWE-287 (Improper Authentication) and CWE-863 (Incorrect Authorization) but even cannot make any prediction for certain CWEs like CWE-918 (Server-Side Request Forgery). The conclusion is that training a general model to fix vulnerabilities automatically is too ambitious to cover all cases. However, if we can focus on specific types of CWEs, the NMT model can provide developers with promising results. SeqTrans can cover about 25% of the types of CWEs in the data set.

The paper makes the following contributions:

1) We use the NMT model transformer to learn and generalize common patterns from historical data for vulnerability fixing.
2) We propose to leverage data-flow dependencies to construct vulnerable sequences and maintain the vital context around them.
3) Fine-tuning has been introduced to overcome the small sample size problem.
4) We implement our approach SeqTrans and evaluate real publicly disclosed vulnerabilities on open-source Java projects. Our SeqTrans outperforms

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

other program repair techniques and achieves the accuracy of 23.3% in statement-level validation and 25.3% in CVE-level validation.

5) We make an internal observation about prediction results on different CWEs and find some interesting CWE fixing operations captured by our model. Our model can predict specific types of CWEs pretty well.

2 Motivating Examples

Figure 1 shows a motivating example of our approach. In Figure 1, there are two vulnerability fixes for CVE-2017-1000390 and CVE-2017-1000388, respectively. These two CVEs belong to the same CWE: CWE-732, which is named "Incorrect Permission Assignment for Critical Resource". CWE-732 emphasizes that "the product specifies permissions for a security-critical resource in a way that allows that resource to be read or modified by unintended actors", which means that when using a critical resource such as a configuration file, the program should carefully check if the resource has insecure permissions.

In Figure 1 (a), before the function `getIconFileName` returns the $IconFileName$, it should check whether the user has the corresponding permission. A similar vulnerability is included in Figure 1 (b). Before the function `EdgeOperation` accesses two resources $JobName$, it should first confirm whether the user has the permission. Otherwise, it will constitute an out-of-bounds permission, which can lead to the leakage of sensitive data such as privacy. Although these two CVEs belong to different projects, their repair processes are very similar. This inspired us that it might be possible to learn common patterns from historical vulnerability fixes that correspond to the same or similar CWEs.

Figure 2 is a more extreme situation, containing two identical CVE modifications CVE-2014-0075 and CVE-2014-0099. These two CVEs belong to the same CWE-189, which is named "Numeric Errors". This CWE is easy to understand. Weaknesses in this category are related to improper calculation or conversion of numbers. These two CVEs contain a series of modifications for overflow evasion, and they are identical. We can directly copy the experience learned in one project to another project.

In this paper, we proposed a novel method to exploit historical vulnerability fix records to provide suggestions and automatically fix the source code. If the function with similar structure requests accesses to a critical resource, our deep learning model can learn to check permissions before allowing access, eliminating the tedious process for developers to search for vulnerability and recapitulate repair patterns.

3 Background

Before describing our approach, we need to briefly introduce the transformer and other tools used in our approach.

Tansformer: In this work, we choose to use the transformer model [46] to solve the performance degradation problem of the seq2seq model on long sequences. It has been widely used by OpenAI and DeepMind in their language models. The implementation of the transformer model comes from an open-source NMT framework OpenNMT [48]. It is designed to be research-friendly to try out new ideas in translation, summary, morphology, and many other domains. Some companies have proven the code to be production-ready.

Unlike Recurrent Neural Network (RNN) [49] or Long Short Term Memory (LSTM) [50] models, transformer relies entirely on the self-attention mechanism to draw global dependencies between input and output data. This model is more parallel and achieves better translation results. The transformer consists of two main components: a set of encoders chained together and a set of decoders chained together. The encode-decoder structure is widely used in NMT models, the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to an embedding representation $z = (z_1, ..., z_n)$, which contains information about the parts of the inputs which are relevant to each other. Given z, the decoder then exploits this incorporated contextual information to generate an output sequence. Generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step, the model consumes the previously generated symbols as additional input when generating the next [51]. The transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder. Each encoder and decoder make use of an attention mechanism to weigh the connections between every input and refer to that information to generate output [46]. The key design of the transformer that brings the biggest performance improvement is to set the distance between any two words to 1, which is very effective in solving the tricky long-term dependency problem in NLP [46].

Fine-tuning: Fine-tuning means taking weights of a trained neural network and using it as initialization or a fixed feature extractor for the task of interest [43]. Why do we need to fine-tune? The reasons are shown as follows [52]:

1) Overcome small sample size: it is impractical to train a large neural network, and overfitting cannot be avoided. At this time, if we still want to use the super feature extraction ability of large neural networks, we can only rely on fine-tuning the already trained models.
2) Low training costs in the later stages: it can reduce training costs and speed up training.
3) No need to build the wheel over and over again: the model trained by the previous work with great effort will be stronger than the model built from scratch in a large probability.

Using this method, we can combine two related works, such as vulnerability fixing and bug repair. The process of fine-tuning usually consists of three parts [52]:

1) Pre-train a neural network model on the source dataset.
2) Create a new neural network target model. It replicates all the model designs and their parameters on the source model except for the last output layer.
3) Train the target model on the target dataset. We will train the output layer from scratch.

```
26 27     public String getIconFileName() {
27    -       return "plugin/jenkins-multijob-plugin/tool32.png";
   28 +       return Jenkins.getInstance().hasPermission(Jenkins.ADMINISTER) ? "plugin/jenkins-multijob-plugin/tool32.png" : null;
28 29     }
```

(a) CVE-2017-1000390, jenkinsci/tikal-multijob-plugin, 2424cec7a099fe4392f052a754fadc28de9f8d86

```
35 35     public EdgeOperation(String sourceJobName, String targetJobName) {
36 36         this.source = Jenkins.getInstance().getItemByFullName(sourceJobName.trim(), AbstractProject.class);
37 37         this.target = Jenkins.getInstance().getItemByFullName(targetJobName, AbstractProject.class);
   38 +         source.checkPermission(Permission.CONFIGURE);
   39 +         target.checkPermission(Permission.CONFIGURE);
```

(b) CVE-2017-1000388, jenkinsci/tikal-multijob-plugin, d442ff671965c279770b28e37dc63a6ab73c0f0e

Fig. 1: Two similar vulnerability fixes belonging to CWE-732

```
322    -        boolean readDigit = false;
   322 +        int readDigit = 0;
344    -                if (charValue != -1) {
345    -                    readDigit = true;
346    -                    result *= 16;
347    -                    result += charValue;
   344 +                if (charValue != -1 && readDigit < 8) {
   345 +                    readDigit++;
   346 +                    result = (result << 4) | charValue;
370    -        if (!readDigit)
   369 +        if (readDigit == 0 || result < 0)
```

(a) CVE-2014-0075, apache/tomcat, f646a5acd5e32d6f5a2d9bf1d94ca66b65477675

```
323    -        boolean readDigit = false;
   323 +        int readDigit = 0;
345    -                if (charValue != -1) {
346    -                    readDigit = true;
347    -                    result *= 16;
348    -                    result += charValue;
   345 +                if (charValue != -1 && readDigit < 8) {
   346 +                    readDigit++;
   347 +                    result = (result << 4) | charValue;
371    -        if (!readDigit)
   370 +        if (readDigit == 0 || result < 0)
```

(b) CVE-2014-0099, apache/tomcat70, 184cdc0d3f03f5737e12d21fff246d7285034597

Fig. 2: Two identical vulnerability fixes belonging to CWE-189

Gumtree: GumTree is the state-of-the-art diff searching tool [53]. It provides several interfaces to accommodate different kinds of parsers such as srcML [54] to parse the source code and build the AST tree. It is worth noting that GumTree only provides a fine-grained mapping between AST nodes, so we modified the code of GumTree and combined it with another tool, Understand [55], to extract the precise diffs. In the meantime, we found some bugs in Gumtree that led to incorrect mismatching and reported them to the author. These issues are explained in more detail in Section 6.2. The algorithm of Gumtree is inspired by the way developers manually look at changes between files. It will traverse the AST tree pairs and compute the mappings in two successive phases:

1) A greedy top-down algorithm to find isomorphic sub-trees of decreasing height. Mappings are established between the nodes of these isomorphic subtrees. They are called anchors mappings.
2) A bottom-up algorithm where two nodes match (called a container mapping) if their descendants (children of the nodes, and their children, and so on) include a large number of common anchors. When two nodes match, an optimal algorithm will be applied to search for additional mappings (called recovery mappings) among their descendants.

4 METHODS

We use the NMT method to automatically guide vulnerability fixing, which aims to learn common change patterns from historical records and apply them to the new input files. In order to overcome the small sample size problem, we introduce the fine-tuning technique. Data-flow dependencies have also been introduced to maintain and capture more critical information around the diff context. SeqTrans can work together with other vulnerability detection tools such as Eclipse Steady [56]. They can provide vulnerability location information at the method level.

4.1 Overview

The overview of our approach is given in Figure 3, which contains three stages: preprocessing, pre-training and fine-tuning, prediction and patching.

Preprocessing: In this step, we will extract diff contexts from two datasets: bug repair and vulnerability fixing datasets. Then, we perform normalization and abstraction based on data-flow dependencies to extract the def-use chains. We believe def-use chains are suitable for deep learning models to capture syntax and structure information around the vulnerabilities with fewer noises. These def-use chains can be fed into the transformer model.

Pre-training and fine-tuning: The training process starts on the bug repair dataset because bug repairs are easier to collect a large enough training set than vulnerability fixes. The tasks of vulnerability and bug fixing have something

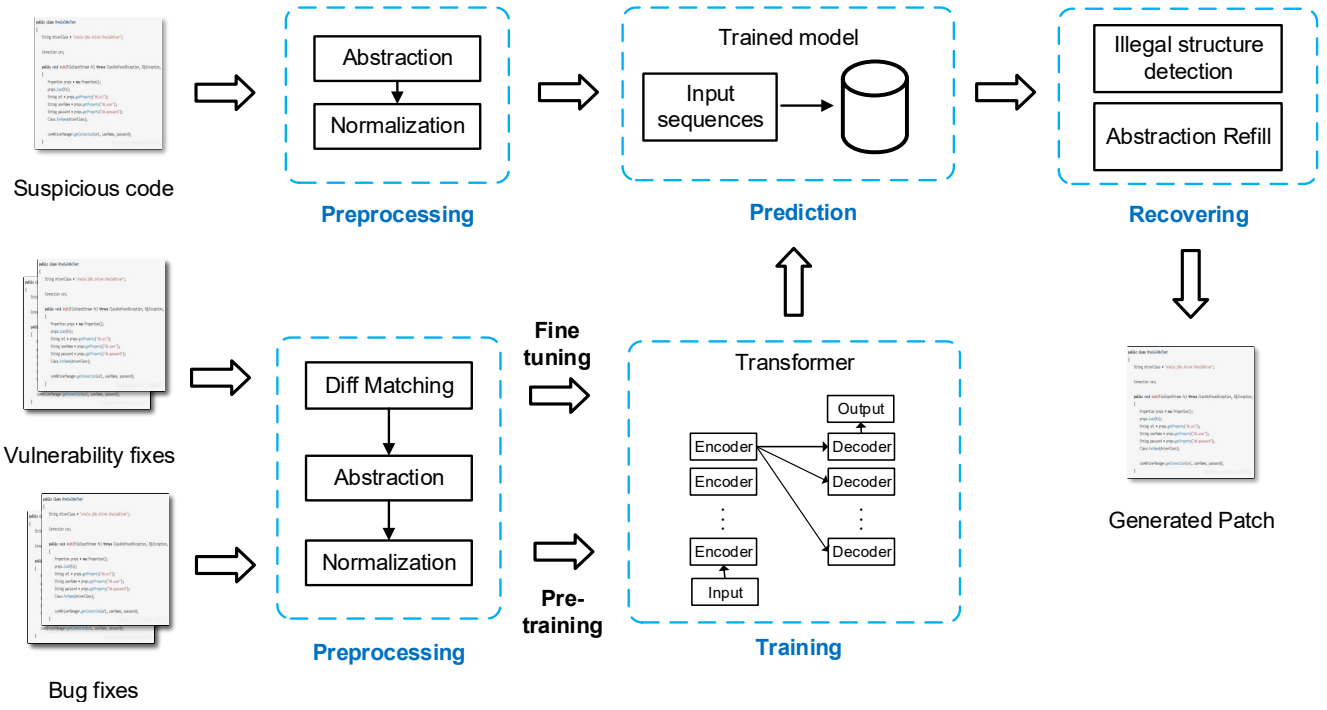


Fig. 3: Overview of our SeqTrans for automatically vulnerability fixing

in common, in other words, something to learn from each other. We can learn and capture parts of general features and hyperparameters from the general task domain dataset, the bug repair dataset. After the pre-training, we will fine-tune the transformer model on the vulnerability fixing dataset. This dataset is much smaller than the first dataset because it is hard to confirm and collect a big enough size for training. Based on the first model, we will refine some of the weights to make the model more suitable for the task of vulnerability fixing. Fine-tuning has been proven to achieve better results on small datasets and speeds up the training process [57], [58].

Prediction and patching: If one vulnerable file is inputted, we need to locate the suspicious codes and predict based on the trained model. In this paper, we do not pay much attention to the vulnerability location part. They can be accomplished by previous vulnerability location tools or with the help of a human security specialist. SeqTrans can provide multiple candidates for users to select the most suitable prediction. Syntax checker Findbugs [59] is exploited to check for errors and filter out predictions that contain syntax errors in advance. After that, we refill abstraction and generate patches. We will discuss the details of each part in the following part of this section.

4.2 Code Change Mining

The two datasets we utilized are Tufano's [39] and Ponta's datasets [40]. Tufano's dataset provides raw source code pairs extracted from the bug-fixing commits, which is easy to be used. However, Ponta's dataset only provides the CSV table containing the vulnerability fixing records. We need a crawler to crawl the project we want. The table contains vulnerability fixing records are shown as follows:

$$(vulnerability\_id; repository\_url; commit\_id)$$

where $vulnerability\_id$ is the identifier of a vulnerability fixed in the $commit\_id$ in the open-source code repository at the $repository\_url$. Each line in the dataset represents a commit that contributes to fixing a vulnerability. Then, we utilize a crawler to collect program repositories mentioned in the dataset. Pull Request (PR) data will be extracted based on $commit\_id$. After that, we need to find out Java file changes involved in each PR. Because our approach SeqTrans only supports Java files now. With the help of a git version control system JGit [60], we can retrieve the version of Java files before and after code changes implemented in the PR. We call these Java file pairs $ChangePair(CP)$, each CP contains a list of code diffs. In some cases, repair operations are performed only on XML or other resource files, or the entire file is refactored directly. In these cases, examples are filtered out. Lastly, we extracted 5K and 650K CPs from Ponta's and Tufano's datasets.

4.3 Code Diff Extraction

After obtaining CPs from PR, we need to locate the diff context. Although we can exploit the *"git diff"* command provided by git to search line-level code diffs, it just does not fulfill our needs. Slight code structure changes such as a new line and adding space are not required. For this reason, we choose to search for code diffs by using Abstract Syntax Trees (ASTs). The state-of-the-art diff searching tool named GumTree [53] is utilized to search for fine-grained AST node mappings.

After that, each CP is represented as a list of code diffs:

$$CP = (st_{src}, st_{dst})_1, ..., (st_{src}, st_{dst})_n$$

where (st_{src}, st_{dst}) represents statements from the source file and the destination file.

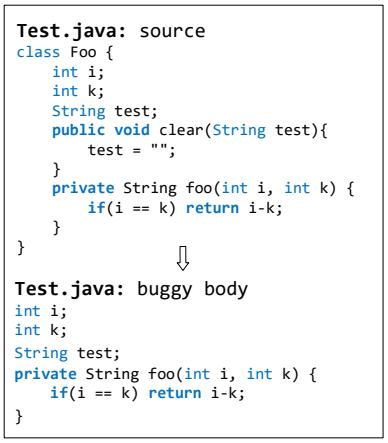


Fig. 4: One example of the buggy body

Then, we will extract data-flow dependencies around code diffs to construct our def-use chains. A def-use chain means assigning some value to a variable, containing all variable definitions from the vulnerable statement. The reasons why we use data-flow dependencies are shown as follows: 1) Context around the vulnerable statements is valuable for understanding risky behavior and capturing structure relationships. However, it is too heavy to maintain the full context with lots of unrelated code at the class level. 2) Data-flow dependencies provide enough context for transformation. If one statement needs to be modified, it is highly likely to co-change its definitions simultaneously. 3) Control flow dependencies often contain branches, making them too long to be tokenized. One example has been given in Figure 4. Assume that the method "foo" contains one vulnerability, we will maintain the method and the vulnerable statement. All global variables will be preserved. All statements that have data dependencies on the vulnerable statement will be retained, too. Statements located after the vulnerable statement within the same method will be removed.

The definition and use (def-use) dependencies can be extracted from the ASTs. The process can be divided into three parts:

1) Traverse the whole AST and label each variable name, constant name, and string name. These names are distributed over the leaf nodes of the AST. This step will be done in the first phase of the modified Gumtree algorithm.
2) Traverse up from the leaf node to search for the defined parent nodes, record the locations.
3) Locate the relevant definition statements of the error-prone statements by location records.

We implement this by modifying the code of Gumtree. Another static analysis tool named Understand is also used to transfer the location records to codes. SeqTrans will change each CP as the following shows:

$$CP = ((def_1, ..., def_n, st_{src}), (def_2, ..., def_m, st_{dst}))_1, ..., ((def_1, ..., def_n, st_{src}), (def_2, ..., def_m, st_{dst}))_n$$

In this paper, we ignore code changes that involve the addition or deletion of entire methods/files.

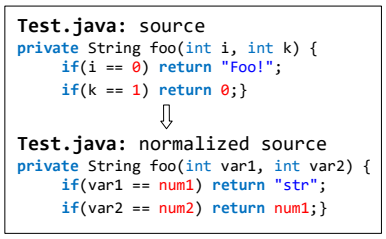


Fig. 5: Normalize the source code

4.4 Normalization & Tokenization

In the training process of the NMT model, there exist a couple of drawbacks. Because NMT models output a probability distribution over words, they can become very slow with many possible words. We need to impose an artificial limit on how many of the most common words we want our model to handle. This is also called vocabulary size. In order to reduce the vocabulary size, we need to preserve the semantic information of the source code while abstracting the context.

The normalization process is shown in Figure 5. We replace variable names to "$var1$", ..., "$varn$", each literal and string are also replaced to "$num1$", ..., "$numn$" and "$liter$". The reasons why we do this involve: 1) reducing the vocabulary size and the frequency of specific tokens; 2) reducing the redundancy of the data and improving the consistency of the data. We will maintain a dictionary to store the mappings between the original label and the substitute to be refilled after prediction. We can control the vocabulary size and make the NMT model concentrate on learning common patterns from different code changes through the above optimization.

Subsequently, we split each abstract CP into a series of tokens. It is worth mentioning that the seq2seq model utilized in previous studies faces severe performance degradation when processing long sequences. For example, Tufano et al.[37] limited the token number to 50-100. By utilizing the transformer model we can better handle long sequences. In our approach, we will limit the CP to 1500 tokens. The vocabulary size is set to 8k based on Gowda's work [61]. We will discuss the details in the following sections.

4.5 Neural Machine Translation Network

In this phase, we train SeqTrans to transform the vulnerable codes and generate multiple prediction candidates. The training process can be divided into two phases: pre-training and fine-tuning.

4.5.1 Pre-training

In the pre-training process, we will utilize a generalized domain corpus from Tufano's dataset for bug repairing to perform the first training. Vulnerability fixing can be considered as a subset of bug repairing. We believe that by pre-training on generic data, we can learn many generic fixing experiences and features that can be applied to the task of vulnerability fixing. A list of $CPs_{general}$ will be extracted by using the approach discussed in section 4.3. These $CPs_{general}$ that contain vulnerable version and fixed version diff context will be given to the network. We will

discuss the network in detail in the following subsection. The pre-training model will be trained for 300K steps till convergence because we found that the validation accuracy smoothed at this training step and no longer fluctuated. In the next fine-tuning process, we will select the model with the highest accuracy on the validation dataset as the final model. The model comes from a breakpoint backup every 5K steps.

4.5.2 Fine-tuning

The purpose of fine-tuning is to improve the model's generalization ability when the target dataset is much smaller than the source dataset. Using this method, we can combine two related works: vulnerability fixing and bug repair. However, one issue is that although fine-tuning is widely used in the Neural Language (NL) field and many pre-training models are provided, there are very few such pre-trained models in the Programming language (PL) field. That is why we need to train the generic domain model by ourselves. The model trained in the previous training process will be fine-tuned using a new vulnerability fixing corpus so that the knowledge learned in the bug repair training can be transferred to the vulnerability fixing task. We set the step size to 1/10 of the pre-training step size. The model selection process is the same as the previous step.

Due to overfitting concerns [62], we will keep earlier layers fixed and only fine-tune the last layer of the model. The training process will update the vocabulary corpus and continue till convergence. A smaller learning rate was selected than the pre-training process, which was set to 0.01. It is worth noting that some studies such as Gururangan's work [63] and documents of OpenNMT[64] mentioned that some sequences were translated poorly (like unidiomatic structure or UNKs) by the retrained model while they are translated better by the base model, which is called "Catastrophic Forgetting". In order to alleviate the catastrophic forgetting, the retraining should be a combination of in-domain and generic data. In this work, we will try to mix part of general domain data into specific domain data to generate such a combination. We have roughly selected some data to be blended into the special domain data on the basis that the blended data should not expand the size of the corpus as much as possible. Eventually, we will double the size of the training set, and the test set will remain unchanged.

4.5.3 Encoder

The encoder is composed of a stack of 6 identical layers. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. Residual connection [65] and normalization [66] have been employed to each sub-layer so that we can represent the output of the sub-layer as:

$$sub\_layer\_output = Layer\_normization(x+(SubLayer(x)))$$

where $Sublayer(x)$ is the function implemented by the sub-layer itself. The self-attention mechanism takes in a set of input encodings from the previous encoder and weighs their relevance to each other to generate a set of output encodings. The feed-forward neural network then further processes each output encoding individually. These output encodings are finally passed to the next encoder as its input. The padding mask has been utilized to ensure that the encoder does not pay any attention to padding tokens. All sub-layers as well as the embedding layers produce outputs of dimension $d_{model} = 512$

4.5.4 Decoder

The decoder also contains a stack of 6 identical layers. However, each layer consists of three sub-layers: an attention sub-layer has been added to perform multi-head attention to draw relevant information from the encodings generated by the encoders. The masking mechanism that contains padding mask and sequence mask has been used to prevent positions from attending to subsequent positions and ensure that the predictions for position i can depend only on the known outputs at positions less than i [46]. The other parts are the same as the encoder.

4.5.5 Attention Mechanism

The purpose of an attention mechanism is to use a set of encodings to incorporate context into a sequence. For each token the attention mechanism requires a query vector Q of dimension d_k, a key vector K of dimension d_k and a value vector V of dimension d_v. These vectors are created by multiplying the embedding by three matrices trained during the training process. The essence of the attention mechanism is actually an addressing process, which is the embodiment of the attention mechanism to alleviate the complexity of the neural network model: instead of feeding all N inputs to the neural network for computation, only some task-relevant information from X needs to be selected and fed to the neural network. Self-attention refers to the situation where the queries, keys, and values are all created using sequence encodings. Then the output Z of this attention mechanism is:

$$Z = Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{n}})V$$

The multi-head attention utilized in the transformer implements several attention mechanisms in parallel and then combines the resulting encoding in a process.

4.6 Prediction and Patch Generation

The original output (or a list of outputs) is far from the version that can be successfully compiled. Because it contains abstraction and normalization, it even may contain grammatical errors after prediction. Our patch generation consists of two steps to solve these problems: abstraction refill and syntax check. In this work, we assume perfect vulnerability localization because different studies may choose different fault localization algorithms, implementations, and granularities such as method-level or statement-level. Liu et al. has pointed out that it is hard to compare different repair techniques due to the reason of different assumptions about the fault localization [67]. We have made a discussion about fault localization in Section 6. Vulnerable codes can come from a classifier, a vulnerability detection tool, or suspicious codes. We will utilize an example from the open-source project called *activemq* to illustrate the process of patch inference and generation.

```
20 ...emq-client/src/main/java/org/apache/activemq/util/ClassLoadingAwareObjectInputStream.java
@@ -34,10 +34,15 @@
34 34      private static final ClassLoader FALLBACK_CLASS_LOADER =
35 35          ClassLoadingAwareObjectInputStream.class.getClassLoader();
36 36
37    -     private static String[] serializablePackages;
   37 +     public static final String[] serializablePackages;
93 89      public static boolean isAllAllowed() {
94    -         return getSerialziablePackages().length == 1 && getSerialziablePackages()[0].equals("*");
   90 +         return serializablePackages.length == 1 && serializablePackages[0].equals("*");
95 91      }
```

Fig. 6: CVE-2015-5254, activemq, 73a0caf758f9e4916783a205c7e422b4db27905c

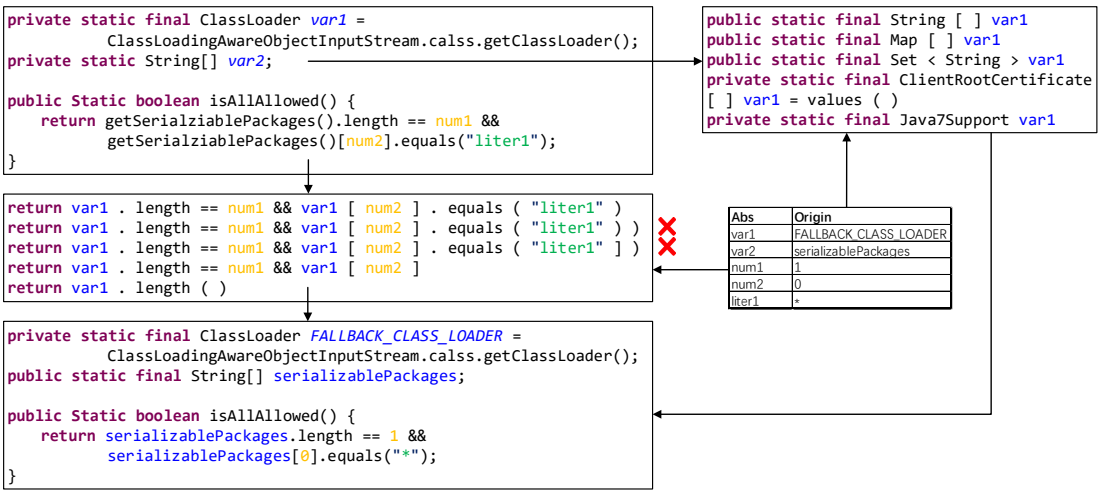


Fig. 7: CVE-2015-5254, activemq, 73a0caf758f9e4916783a205c7e422b4db27905c

Figure 6 shows a CVE repair record in *activemq*, which contains three statement fixes. Firstly, as mentioned in Figure 3, the input codes need to be abstracted and normalized. We decompose them into sequences following a similar process as depicted in Figure 7. In Figure 7, every abstracted variable has been marked in blue color, with every constant in yellow color and every literal in green color. Each sequence will maintain a dictionary for future recovery. The location of the sequence will also be recorded for subsequent backfill. Then, these sequences are fed into the transformer model, beam search [37] are used to generate multiple predictions for the same vulnerable statement. The output of the network is also abstracted sequences like Figure 7. It is a sequence that contains the predicted statement and the context around it. Thirdly, we backfill all the abstractions when a prediction is selected and apply syntax checks. The next subsections will supplement some concrete techniques and tools applied in this process.

4.6.1 Beam Search

In many cases, developers have specific domain-specific knowledge. We can generate a list of prediction results to let developers pick the most suitable one. Beam search a heuristic graph search algorithm [68], [69]. Instead of greedily choosing the most likely next step as the sequence is constructed, the beam search expands all possible next steps and keeps the k most likely, where k is a user-specified parameter and controls the number of beams or parallel searches through the sequence of probabilities. Beam search maintains the n best sequences until the upper limit of the set beam size.

As has been depicted in Figure 7, each of the vulnerable statements will generate five prediction candidates. Usually, the highest-ranked predictions will be chosen and utilized. In some cases, there are syntax errors in the prediction results. We will use syntax checking tools to detect these errors. This will be discussed in the following subsections These k candidates will be provided as suggestions to developers to select the best result.

4.6.2 Abstraction Refill

As has been shown in Figure 7, SeqTrans will maintain a dictionary to store the necessary information for restoration before abstraction. After prediction, the output will be concretized, and all the abstractions in the dictionary will be refilled. The code will be automatically indented in this process. It should be noted that all comments will be deleted and will not be refilled again. The dictionary we maintain will store relevant variable, constant and literal for the whole CP. We believe that the search space explosion is

not an important issue at this scale. One shortcoming of SeqTrans is that the mappings included in the dictionary come from the source files. If the abstraction is the content that needs to be repaired, it is hard for SeqTrans to understand and infer them. All we can do is reduce the corresponding abstraction according to the dictionary. For example, if one *println* function changes what it wants to print. The model has difficulty predicting what it wants to print. If a predicted abstraction cannot find a mapping in the dictionary, we will copy the original abstraction content to the current location.

4.6.3 Syntax Check

We combine beam search with a grammar check tool to analyze the syntax and grammatical errors contained in the predictions. The static analysis tool *FindBugs* [59] is exploited to identify different potential bugs in Java programs. The version we utilized is 3.0.1. The motivation for introducing static analysis is to filter out as many invalid generation patches as possible before executing test cases. Because the time cost of running all the test cases is very high. Potential errors can be divided into four levels: scariest, scary, troubling, and of concern based on their possible impact or severity.

In SeqTrans, one generated patch needs to pass the compiler first and then the FindBugs detection. If the candidate prediction cannot pass the checking process, it will be filtered. It should be noted that Findbugs may trigger a warning even on the pre-commit version, so we only check the warning messages that are added after the prediction. For example, in Figure 7, the second and the third candidates contain a syntax error, which cannot pass the check of FindBugs. We will remove these two candidates. In other words, we use FindBugs to check the candidates to ensure that the five candidates we recommend introduce as few new bugs as possible. We also make an evaluation for this checker in the experimental Section.

Finally, we can generate the newly patched file and provide it to developers. We provide flexible choices for developers to enable this feature or judge by their domain-specific knowledge. Developers also have the flexibility to choose the predictions they need based on their own domain experience and based on our five recommended candidates. In addition, we believe that with the continuous improvement of model training, these grammatical errors will become less and less. In the end, we will no longer rely on third-party grammatical error check tools.

5 Empirical Study & Evaluation

In this section, we conduct our experiment on a public dataset [40] of vulnerability fixes and evaluate our method: SeqTrans by investigating three research questions.

5.1 Research Questions

We explore the following research questions:

- **RQ1:** How much effectiveness can SeqTrans provide for vulnerable code prediction?
  RQ1 aims to prove that the NMT-based technique is a feasible approach to learn automated code transformations, and SeqTrans outperforms other state-of-the-art techniques.
- **RQ2:** What are the characteristics of the ML model used that can impact the performance of SeqTrans.
  RQ2 will evaluate the impacts of the main components of SeqTrans on performance, such as the data structure and the transformer model.
- **RQ3:** How does SeqTrans perform in predicting specific types of CWEs?
  RQ3 will explore in-depth the prediction results and the source codes of the data set to observe whether our method performs inconsistently when predicting different kinds of CWEs.

5.2 Experimental Design

In this section, we discuss our experimental design for RQ1, RQ2, and RQ3. All experiments were accomplished on a server with an Intel Xeon E5 processor, four Nvidia 3090 GPU, and 1TB RAM.

Dataset: Our evaluation is based on two public datasets: Tufano's [39] [1] and Ponta's datasets [40] [2]. Tufano's dataset contains 780,000 bug fix commits and nearly 2 million sentence pairs of historical bug fix records. For each bug-fixing commit, they extracted the source code before and after the bug-fix using the GitHub Compare API [70]. Each bug-fixing record contains the buggy (pre-commit) and the fixed (post-commit) code. They discarded commits related to non-Java files and new files created in the bug-fixing commit since there would be no buggy version to learn. Moreover, they discarded commits impacting more than five Java files since they aim to learn focused bug fixes that are not spread across the system.

Ponta's dataset was obtained from the National Vulnerability Database (NVD) and from project-specific Web resources that they continuously monitor. From that data, they extracted a dataset that maps 624 publicly disclosed vulnerabilities affecting 205 distinct open-source Java projects, used in SAP products or internal tools, onto the 1282 commits that fix them. The distribution of these CVEs ranges from 2008 through 2019. Out of 624 vulnerabilities, 29 do not have a CVE identifier, and 46, which do have a CVE identifier assigned by a numbering authority, are not available in the NVD yet. These vulnerabilities have been removed from the dataset, the final number of non-repetitive CVEs is 549 with 1068 related commits. In total, the processed Ponta's dataset contains 1068 different vulnerabilities fixing commits with 5K diff contexts across 205 projects classified as 77 CWEs from 2008 to 2019. Figure 8 shows the CWE distribution in descending order of frequency, with the cumulative yellow line on the secondary axis, identifying the percentage of the total number. In the appendix, we have listed the IDs and type explanations of all CWEs in Ponta's dataset.

The datasets are released under an open-source license, together with supporting scripts that allow researchers to automatically retrieve the actual content of the commits from the corresponding repositories and augment the attributes available for each instance. Also, these scripts complement the dataset with additional instances that are not

1. https://sites.google.com/view/learning-fixes/data
2. https://github.com/SAP/vulnerability-assessment-kb

security fixes (which is useful, for example, in machine learning applications).

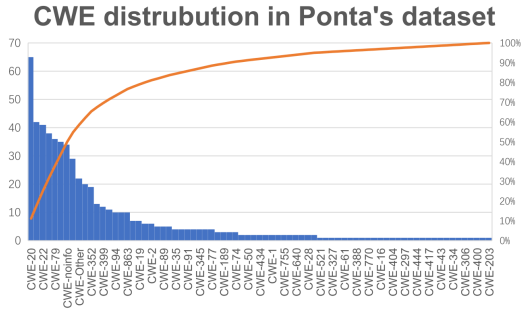


Fig. 8: CWE distribution of Ponta's dataset

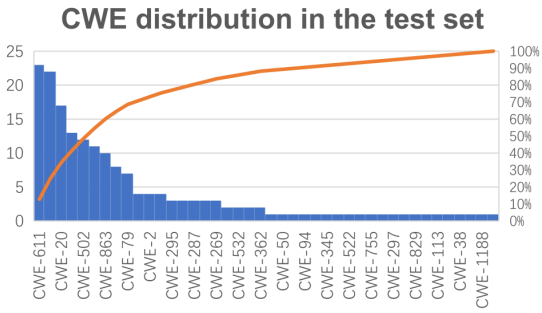
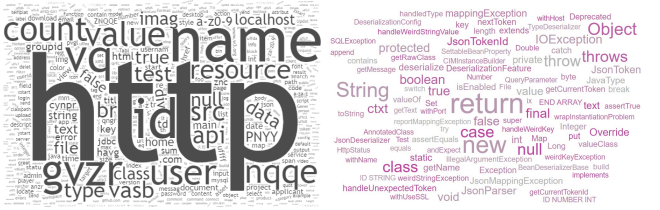
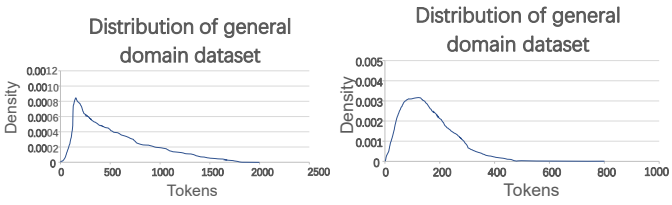


Fig. 9: CWE distribution of the test set



(a) Tufano's dataset (b) Ponta's dataset

Fig. 10: Label distribution for each dataset



(a) Tufano's dataset (b) Ponta's dataset

Fig. 11: Token distribution for each dataset

Validation: We will use three methods to validate the performance of the experiments.

The first validation set T_{cross} is 10-fold cross-validation. Cross-validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model and a test set to evaluate it. In 10-fold cross-validation, the original sample is randomly partitioned into ten equal size subsamples. Of the ten subsamples, a single subsample is retained as the validation data for testing the model, and the remaining nine subsamples are used as training data. The process is then repeated ten times (the folds), with each of the ten subsamples used exactly once as the validation data. It should be noted that cross-validation is only applied to fine-tuning process. All nine subsamples will share the same pre-training set. If the predicted statement equals the statement in the test set, there is a correct prediction. The ten results from the folds will be averaged to produce a single estimation. The advantage of this method is that each sample of data is used as training data and test data. The over-learning and under-learning states are avoided, the results obtained are more convincing.

The second validation set T_{cwe} is based on the chronological relationship of the CVE repair records to simulate the actual development process of using historical vulnerability fix records to fix subsequent suspicious code. We also sorted the CVE samples in Ponta's dataset by time series and used the CVE fix records from 2008 to 2017 as the training set (708 CPs), the CVE fix records from 2018 and 2019 were utilized as the validation (136 CPs) and test sets (150 CPs). If one CP has been fully and correctly predicted, we regard it as one successful fix. The distribution of the 42 CWEs in the test set is shown in Figure 9. The previous two validations do not contain compilation and syntax checker in the abstraction refill part. We match the refilled statements strictly with the statements in historical repair records to verify if it is a correct patch. We will verify the performance of this part in the next validation experiment.

TABLE 1: Detailed information of T_{tra}

Project Name	CVE Number	CWE Number
UAA	37	10
Struts	30	8
Spring-framework	26	11
Lucene-solr	14	2
Jenkins	13	8

Figure 10 shows the label distribution of each dataset. We can find that the frequency distribution of labels in the two datasets is very dissimilar. Figure 11 shows the token distribution of the vulnerable abstract context in each dataset. It should be noted that the token length that is bigger than 2000 has been ignored in Tufano's dataset. The token length bigger than 800 has been ignored in Ponta's dataset. We can find that most tokens in Tufano's dataset are distributed between 0 and 1500. The majority of tokens in Ponta's dataset are distributed between 0 and 400.

In the third test set T_{tra}, we will try to use traditional evaluation approaches. Five open-source projects which contain the largest number of CVEs (120 CPs) will be selected as the test sets. Fixing records on other projects will be used as training sets. We will input suspicious files into the model of SeqTrans to generate patches. We consider it a

successful prediction if the predicted file passes the relevant test case and no new failures are introduced. The detailed information of the test sets is shown in Table 1.

In Table 1, the first column shows the project name, including CloudFoundry User Account and Authentication Server (UAA), Apache Struts, Spring framework, Apache Solr and Jenkins. Except for Apache Solr, every one of them has received more than 1K stars on Github. Each of them has more than ten years of development history and has a stable maintenance team. We believe that their CVE fix records are relatively reliable and follow the specifications. The second column shows the number of CVEs included in each project, and the third column shows the number of CWEs contained in each project. It should be noted that nearly 5% of the commit records were removed because they failed to pass compilation or the version was too old. In addition, because these projects have long maintenance cycles and use different version control tools and development environments. We manually configured all remaining project versions to ensure that each one would compile successfully and pass as many test cases as possible.

5.2.1 ***RQ1 Setup***:

The experimental part of RQ1 will be divided into three components, RQ1.1, RQ1.2 and RQ1.3.

Firstly, RQ1.1 will show and analyze the joint training and independent training results of the two datasets. Since SeqTrans uses two datasets and a fine-tuning approach to overcome the problem of small samples, then independent and joint analyses for both datasets are necessary. For the bug repair dataset of the general domain, we will train on G_{train} and validate on G_{val}. G_{val} is separated from the bug repair dataset, which is not contained in G_{train}. Likewise, we will separate the vulnerability dataset of specific domain to S_{train}, S_{val} and S_{test}. The S_{test} will be utilized to validate the performance for both joint training and independent training. Sequences in each set are mutually exclusive. This experiment is designed to verify whether fine-tuning can help small samples overcome the problem of dataset size, learn from general domain tasks, and transfer it to specific domain tasks.

Secondly, RQ1.2 will compare SeqTrans with some state-of-the-art techniques such as Tufano [37], [71] et al. and SequenceR [38]. In order to avoid the effects of using pre-trained models, we will divide SeqTrans into SeqTrans_full and SeqTrans_single to refer to methods that use the pre-train model and the one that do not use the pretrain model. SeqTrans_full can be regarded as an enhancement of SeqTrans_single as to alleviate the overfitting problem. In the following sections, all SeqTrans that are not specified refer to SeqTrans_full.

Tufano has investigated the feasibility of using neural machine translation for learning wild code. The disadvantage of his method is that only sentences with less than 100 tokens are analyzed. SequenceR presents a novel end-to-end approach to program repair based on sequence-to-sequence learning. It utilizes the copy mechanism to overcome the unlimited vocabulary problem. To the best of our knowledge, it achieves the best result reported on such a task. However, the abstract data structure of this method retains too much useless context. It does not use the normalization method either. We have also added the model that utilizes the same data structure but uses the seq2seq model. Seq2seq model is an RNN encoder-decoder model widely used in the NMT domain, previous approaches such as SequenceR [38] and Tufano et al. [37] is also based on this model. We have calculated the prediction accuracy for each technique. Prediction accuracy will be calculated using 10-fold cross-validation for each technique. Then we will calculate the number of correct predictions divided by the total number to calculate the accuracy.

Thirdly, RQ1.3 will apply SeqTrans on T_{tra}, the five projects selected from Ponta's dataset with the traditional evaluation approach. Suspicious files will be input to the fine-tuned SeqTrans model to generate multiple patches. The beam size is set to 10 but not 50 because it takes too long to compile and complete the test process. The predicted and restored files will be sent back to the project to overwrite the source files. Then, we will recompile the whole project and run the test cases. There is a vital problem here, how to define a vulnerability is successfully fixed? We will manually search and compare the parent commit of this CVE fix record. If predicted files are compilable, all the diffs are semantically modified, and no new test failures are introduced, we consider it a correct fix.

Generated patches will be categorized into three types:

- Compilable: The patch can pass the compiler.
- Plausible: The patch can pass the compiler and the test suite.
- Correct: The patch can pass the compiler and the test suite. It has also passed our manual checking.

These three types are inclusive relationships. If the modified statement matches the changes in the commit, we consider it to be a correct patch. If the modified statement does not match the changes in the commit, it will be manually determined if it affects the code logic. The plausible patches are manually checked by the first and the second author of this paper. Both of them have more than five years of Java development experience.

5.2.2 ***RQ2 Setup***:

In this part, we will discuss the impacts of the main factors that affect the performance of SeqTrans.

The process is shown as follows: Firstly, we will select a list of parameters that may affect the performance of our model. Then we will change one parameter at one time and make the experiment in the same dataset. We will utilize cross-validation ten times for each parameter and calculate the mean value as the final precision. The final parameter selections of SeqTrans will produce the highest acceptance rates for the alternative configurations and data formats we tested.

5.2.3 ***RQ3 Setup***:

In this part, we will discuss the observations when we look deep inside the prediction results. We only manually analyzed the prediction results generated by SeqTrans. Other models are not considered.

We have calculated the prediction accuracy for each CWE and each category of code transformation. We will look deep inside some well-predicted CWEs to explore why

SeqTrans performs better on them. We will also analyze why some CWEs have very poor prediction performance.

5.3 Experimental Results

5.3.1 RQ1: How much effectiveness can SeqTrans provide for vulnerable code prediction?

In RQ1, our goal is to analyze the performance of SeqTrans on the task of vulnerability fix. As we have mentioned before, RQ1 will be divided into three components. Firstly, we will analyze the joint training and independent training results of the two datasets in RQ1.1. Table 2 shows the prediction accuracy of models which were trained only on the general domain dataset (only on Tufano's dataset) or trained only on a specific domain dataset (only on Ponta's dataset) or trained jointly (fine-tuning strategy). The first column is the training approach of the three models. The second column is the beam search size. For example, in the situation of Beam=10, for each vulnerable sequence, we will generate ten prediction candidates. If one of these ten candidates contains the correct prediction, the prediction accuracy is 1 otherwise it is 0. The third column is the total prediction accuracy. Recall that we use 10-fold cross-validation to calculate the accuracy of the model. If the predicted statement equals the statement in the test set, there is a correct prediction.

RQ1.1: From Table 2, we can observe that SeqTrans that use the fine-tuning strategy achieves the best performance of 14.1% when Beam=1 and 23.3% when Beam=50. Next is the performance of 11.3% when Beam=1 and 22.1% when Beam=50 achieved by training on a specific domain dataset. The worst prediction performance is using only data sets from the general domain, it can just achieve the accuracy of 4.7% when Beam=1 and 6.9% when Beam=50. Detailed Beam search results are shown in Figure 14 when beam size increases from 1 to 50. The x-axis represents beam size and the y-axis represents the prediction accuracy.

Results show that using fine-tuning strategy to transfer knowledge from the general domain of bug repairing to the specific domain of vulnerability fixing improved the prediction performance of SeqTrans and achieved better performance than doing the training on two separate datasets. Fine-tuning is helpful to alleviate and overcome the small data size problem. In the following experiments, the fine-tuning strategy will become one of the default configurations in SeqTrans.

RQ1.2: Secondly, we will compare SeqTrans with some state-of-the-art techniques. Table 4 shows the accuracy results of single line prediction in five different NMT models including SeqTrans_full, SeqTrans_single, Seq2seq model, SequenceR, and the work of Tufano et al.. SeqTrans_full, SeqTrans_single refer to SeqTrans models that have been pre-trained and fine-tuned, and SeqTrans models that have been trained using only the Ponta's dataset. For the Seq2seq model and transformer model, we use the same training set with def-use chains. As for the SequenceR [38] and Tufano et al. [71], we will strictly follow their original codes and data structures, repeat their preprocessing, training, and translating steps.

The reason why the total number in T_{cross} is inconsistent is that the data structure in different approaches is not the same. SequenceR packages the entire class containing the buggy line, keeps the buggy method, all the instance variables, and only constructor's signature and non-buggy methods (stripping out the body). Then it performs tokenization and truncation to create the abstract buggy context. Because this abstract buggy context maintains too much context, even the whole buggy method and the constructor's signature in the class have the highest total number after deduplication. Tufano et al. only construct the buggy pair that contains the buggy method and the corresponding fixed method. However, they limit the whole sentence to 100 tokens and do not contain any statement outside of the method, so that this approach has the lowest total number after deduplication. As introduced in Section 4, our approach will maintain the buggy method with the vulnerable statement and any statement that has a data dependency on the vulnerable statement. The total number of our approach is in the middle.

In order to maintain a relatively fair training and testing environment, we introduce a second verification method. As it has been explained previously, T_{cwe} provides an identical set of raw training, validation, and test dataset for each approach. If one CP has been fully and correctly predicted, we regard it as a successful fix. We have also tried to exploit the beam search to generate a list of predictions. Figure 15 shows the performance on T_{cross} when beam size increases from 1 to 50. The x-axis represents beam size and the y-axis represents the prediction accuracy.

From table 4, we see that our SeqTrans_full performs the best and achieves an accuracy of 301/2130 (14.1%) when Beam=1 on T_{cross}, followed by SeqTrans_single 338/2130 (11.3%), Seq2seq 121/2130 (7.5%), SequenceR 252/3661 (6.9%) and Tufano et al. 37/883 (4.2%). On T_{cwe}, SeqTrans_full also reaches the best accuracy of 35/150(23.3%) when Beam=1, followed by SeqTrans_single 26/150 (17.3%) SequenceR 24/150 (16.0%), Seq2seq 20/150 (13.3%) and Tufano et al. 5/150 (3.3%). The experimental results of T_{cross} and T_{cwe} are generally consistent. We will do a more detailed case study in the RQ3.

To our surprise is that SequenceR is not as good as described. It even performs worse than Seq2seq when Beam=1 on T_{cross}. The difference between data structures can explain the poor performance of SequenceR. SequenceR utilizes the buggy context, which contains the buggy line and the context around the buggy line in the same function. Other variable and method declarations in the same class will also be retained. However, this buggy context keeps many statements with no relationship with the buggy line. The whole data structure is too long and contains numerous declaration statements unrelated to the buggy line, which performs poorly in our vulnerable public dataset. Another disadvantage is that SequenceR only supports single-line prediction, but there are cases of statement deletions and additions included in the vulnerability fix.

In our SeqTrans, we only maintain the data dependencies before the vulnerable statement. Meanwhile, we will normalize the data and replace variable names by "$var1, var2....vark$". The literals and numerical values will also be replaced by constants and maintained in a dictionary

for future recovery. The poor performance of Tufano et al. may be due to few data samples. We strictly follow their method and only select sequences with less than 100 tokens. On the other hand, the fine-tuning method we use to learn from the general domain improves performance. Another observation is that setting the beam size to 10 is sufficient in most cases. Overall, SeqTrans leverages def-use chains and fine-tuning strategy to maintain data dependencies and overcome the minor data size issue, which can help the NMT model reach higher accuracy.

RQ1.3: Thirdly, we will use our SeqTrans to perform a traditional evaluation on five open source projects which contain the largest number of CVEs. Table 3 shows the results of these five projects. The first column is the project name and the second column is the overall number. The third column is the compilable number, which means that at least one of the patches in this commit version is compilable. The fourth column is the plausible number, it requires that the patch not only be compilable but also pass the test suite. The fifth column is the number of correct patches, we will manually check the plausible patches to ensure these changes are semantically and functionally equivalent to the historical fixes.

Results show that out of 120 vulnerabilities, SeqTrans generates at least one compilable patch for 98 vulnerabilities. Some suspicious files cannot generate one compilable patch because some fixed records add or remove entire methods or rewrite the entire file. For example, in *SECURITY-499* of Jenkins, it rewrites two files and the associated test cases. This case cannot be correctly fixed by our approach now. SeqTrans also generates at least one plausible patch for 30 vulnerabilities. This number is much smaller than the compilable number because many fixing histories not only modify source files but also change resource files such as the configuration files. For another case, one fix may introduce new third-party packages. This situation cannot be fixed by our approach now. Finally, SeqTrans successfully generated at least one correct patch for 21 vulnerabilities. We can see that nearly 18% of the 120 vulnerabilities are fixed. These patches have been manually checked to ensure they are semantically equivalent to the historical fixing records. Figure 12 shows a fixing fragment of CVE-2016-0785, CWE-20 in Struts. There is a pair of useless brackets in the prediction results of SeqTrans (the third line). However, it does not influence the function of the statement. In this case, we also treat it as a correct fix.

Figure 13 shows a global statistic for T_{tra}. In the figure, we add a checked tag to analyze the effectiveness of FindBugs, which means the patch that has passed the static analysis check. For a total of 1200 generated patches, 438 patches can be compiled. Then, after the checking of FindBugs, 413 patches are survived. In these patches, 49 of them are plausible, and finally, 25 patches are validated to be correct, which means they have passed the relevant test cases and are semantically equivalent to historical fix records. Here we give some observations from Figure 13. The compiler filters out most of the 787 invalid patches filtered by the compiler and the checker. The checker only filtered out 25 patches. FindBugs actually reports more numbers than this, but most of them are not associated with the vulnerable statements. The total plausible number in Figure 13 is larger than Tabel 3, which means there is more than one plausible patch for one CVE. This situation can heavily rely on the quality of the related test cases [72]. This gap will be reduced if the developer commits the relevant test set changes together with the commit promptly. This result makes us consider whether removing the checking part to reduce the overhead is good. We will explore more options in our future work.

In general, the current functionality of SeqTrans is suitable as assistance to developers for program repair. There is still a long way to separate from the developers and independently do accurate automatic program fixes.

TABLE 2: Prediction results in three training strategies

Approach	Beam	Accuracy
Only on general domain G_{train}	1	100/2130(4.7%)
	10	121/2130(5.7%)
	50	146/2130(6.9%)
Only on specific domain S_{train}	1	242/2130(11.3%)
	10	338/2130(15.5%)
	50	473/2130(22.1%)
Joint training on G_{train} and S_{train}	1	301/2130(14.1%)
	10	411/2130(19.3%)
	50	497/2130(23.3%)

TABLE 3: Effectiveness on the five selected projects

Project Name	Total	Compilable	Plausible	Correct
UAA	37	31	9	6
Struts	30	25	10	7
Spring-framework	26	21	6	4
Lucene-solr	14	11	3	2
Jenkins	13	9	2	2

TABLE 4: Performance of different techniques

Approach	Beam	Accuracy	
		T_{cross}	T_{cwe}
SeqTrans_full	1	301/2130(14.1%)	35/150(23.3%)
	10	411/2130(19.3%)	38/150(25.3%)
	50	497/2130(23.3%)	38/150(25.3%)
SeqTrans_single	1	242/2130(11.3%)	26/150(17.3%)
	10	338/2130(15.5%)	31/150(20.7%)
	50	473/2130(22.1%)	31/150(20.7%)
SequenceR	1	252/3660(6.9%)	24/150(16.0%)
	10	418/3660(11.4%)	26/150(17.3%)
	50	725/3660(19.8%)	27/150(18.0%)
Seq2seq	1	121/2130(7.5%)	20/150(13.3%)
	10	242/2130(11.3%)	23/150(15.3%)
	50	390/2130(18.3%)	23/150(15.3%)
Tufano et al.	1	37/883(4.2%)	5/150(3.3%)
	10	59/883(6.7%)	7/150(4.6%)
	50	63/883(7.1%)	7/150(4.6%)

```
> if (ComponentUtils.altSyntax(getStack())&&ComponentUtils.isExpression(value)){
= if (ComponentUtils.altSyntax(getStack())&&ComponentUtils.isExpression(value.toString())){
> if (ComponentUtils.altSyntax(getStack())&&ComponentUtils.isExpression((value.toString()))){
```

Fig. 12: Case: fixing snippet of CVE-2016-0785 in Struts

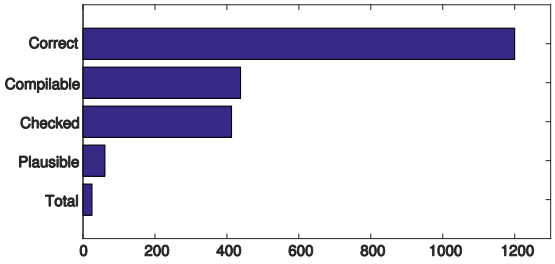


Fig. 13: Statistics of SeqTrans generated patches for T_{tra}

Answer to RQ1: In summary, NMT models are able to learn meaningful code changes from historical code repair records and generate predicted patch to assist developers with code repairs. Our approach SeqTrans based on a transformer model outperforms other NMT models on the task of vulnerability fixing. Even it outperforms the state-of-the-art approach SequenceR in our public vulnerability fix dataset.

5.3.2 RQ2: What are the characteristics of the ML model used that can impact the performance of SeqTrans?

In RQ2, we will discuss some of the data formats and configuration exploration processes that we have tried to get a default SeqTrans model eventually. Table 5 and Figure 16 shows an ablation study for SeqTrans. From Table 5, we can see the prediction result of our default SeqTrans against the results of single changes on the model. We will explain them one by one. These ablation results will help future researchers understand which configurations are most likely to improve their own models. Due to the random nature of the learning process, we will use the 10-fold cross-validation on T_{cross} to train each control group 10 times and take the mean value as the final result. The first row is the performance of the default SeqTrans model as a reference.

Group 1 in the second and third rows explored the effect of word size on the performance of our model. Results show that both the smaller and larger word sizes perform worse than the configuration we choose. We think the reason is that Smaller word sizes may lead to transitional compression of features and loss of some valid information. Larger word sizes may not be appropriate for the size of our dataset.

In Group 2 and Figure 16b we have discussed whether more training steps would significantly improve performance. The result indicates that the performance difference between 30K and 100K training steps is very small. The growth in predicted performance begins to converge after 30k training steps. We do not consider it worthwhile due to the large time overhead of 100K training steps. It is worth noting that the training step here refers to the step used when fine-tuning the dataset of vulnerability fixing tasks in the special domain, and the general domain model is consistent.

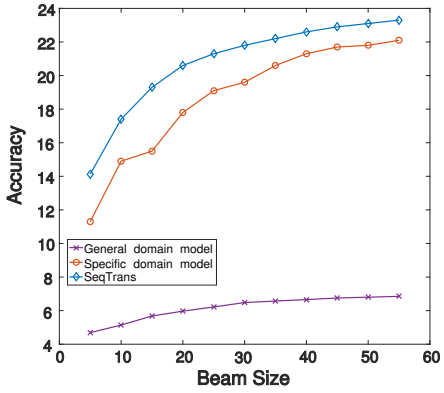


Fig. 14: Performance of three training strategies

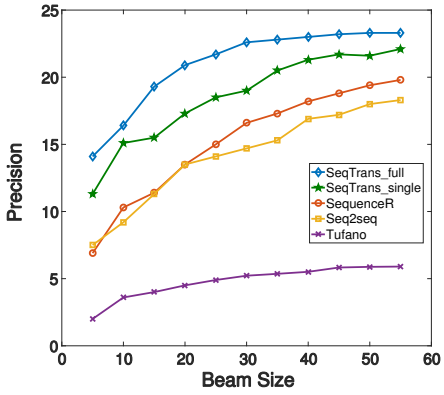
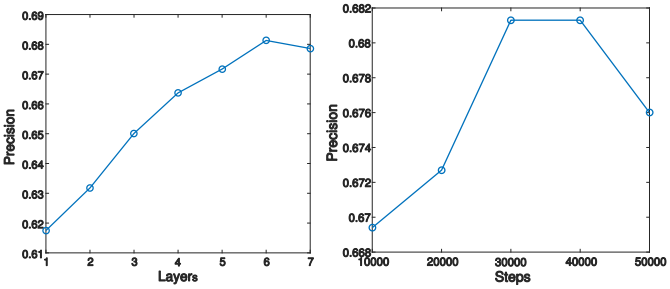


Fig. 15: Performance of different techniques



(a) Layers (b) Training Steps

Fig. 16: Factor analysis with selected parameters

Group 3 in the fifth and sixth rows and Figure 16a are the test of model layers, we have tried different features and the conclusion is that 6 layers are a suitable choice. It is worth noting that we need to ensure that the encoder and decoder parts of the transformer model have the same number of layers, so we use the same number of layers on both the encoder and decoder. Results show that prediction performance rises with the number of layers until it reaches 6. The performance of layer 7 is not better than 6, so we decide on 6 as the parameter. Group 4 and Group 5 are the test of different batch sizes and hidden state sizes. The experimental results show a similar conclusion: decreasing the size leads to decreased performance.

In group 6, 7 and 8, we will discuss the impact of data structure and processing on performance. The result

TABLE 5: Factor impact analysis with selected parameters

Group	Description	Precision	Impact
-	Default SeqTrans model	23.3%	-
1	Word Size (256 vs 512)	22.4%	-4%
	Word Size (512 vs 1024)	22.1%	-5%
2	Training steps (30K vs 100K)	23.5%	1%
3	Layers (5 vs 6)	21.9%	-6%
	Layers (6 vs 7)	22.4%	-4%
4	Batch Size (2048 vs 4096)	22.6%	-3%
5	Hidden State Size (256 vs 512)	22.8%	-2%
6	Without Def-use Chains	20.9%	-10%
7	Without Code Normalization	21.9%	-6%
8	Without BPE	23.3%	0%
9	Without Mixed Fine-tuning	22.1%	-5%
10	Without Fine-tuning Strategy	20.2%	-13%

shows a 10% improvement in model performance when comparing our data structure to the original single vulnerable line. Normalization in data preprocessing will lead to a 6% increase in performance. An interesting phenomenon is that whether BPE is enabled or not has only a minimal performance impact. We think the main purpose of BPE is to compress the data and solve the problem of unregistered words. Our vocabulary size is able to cover the majority of words. However, when we prepare the first general model, not using BPE to compress the sequences will cause a huge vocabulary size and lead to the overflow of GPU memory.

Group 9 is designed to explore whether mixing some general domain training data into the small specific domain dataset can alleviate the problem of catastrophic forgetting. We tried to mix in the same number of randomly selected G_{train} training data as S_{train} and compare the results with the original S_{train} experiments. The result shows that without mixing the prediction performance indeed causes a degradation of the performance. The last Group 10 is the performance change before and after using the fine-tuning strategy as explained in the previous experiments. SeqTrans achieves a 13% performance improvement, indicating that the fine-tuning strategy is very beneficial for training small-scale data and helps us migrate knowledge from similar domains.

Answer to RQ2: The ablation study results demonstrate that parameter selections for the SeqTrans produce the highest acceptance rates for the configurations we tested. These ablation results will help future researchers understand which configurations are most likely to improve their own models.

5.3.3 RQ3: How does SeqTrans perform in predicting specific types of CWEs?

We now look at what types of vulnerabilities fix our model can well identify and generate predictions. The purpose of this experiment is to verify whether SeqTrans has better performance for a specific type of CWE. For example, the CWEs have a high number of repair cases in the dataset or the CWEs are uniformly distributed in the data set by time

TABLE 6: Prediction results in the data set

T_{cross}			T_{cwe}		
CWE No.	**Accu**		**CWE No.**	**Accu**	
CWE-444	3/5	0.60	CWE-306	1/1	1.00
CWE-287	45/84	0.54	CWE-287	2/3	0.67
CWE-306	1/2	0.50	CWE-20	8/14	0.57
CWE-362	5/11	0.45	CWE-522	2/4	0.50
CWE-22	13/30	0.43	CWE-22	10/21	0.48
CWE-361	3/7	0.43	CWE-295	1/3	0.33
CWE-863	7/17	0.41	CWE-269	1/3	0.33
CWE-284	3/8	0.38	CWE-863	3/10	0.30
CWE-522	24/67	0.36	CWE-502	5/12	0.42
CWE-20	31/97	0.32	CWE-611	3/13	0.23
CWE-502	311/1013	0.31	CWE-200	2/11	0.18
CWE-78	7/23	0.30	CWE-noinfo	2/13	0.15
CWE-74	4/14	0.29	CWE-78	0/5	0
CWE-310	41/147	0.28	CWE-35	0/3	0
CWE-269	8/29	0.28	CWE-601	0/2	0
CWE-264	14/60	0.23	CWE-74	0/2	0
CWE-611	1/52	0.21	CWE-362	0/1	0
CWE-noinfo	7/54	0.13	CWE-521	0/1	0
CWE-200	3/28	0.11	CWE-50	0/1	0
CWE-19	5/56	0.09	CWE-89	0/1	0
All	563/2130	26.4%	All	40/150	26.7%

series. Table 6 shows the prediction accuracy of each CWE in T_{cross} and T_{cwe} when Beam=50. The Common Weakness Enumeration (CWE) is a category system for software weaknesses and vulnerabilities. Every CWE contains a list of CVEs. Because there are too many kinds of CWE, we only list the top 20 with the highest accuracy in the table, which contains the vast majority of correct predictions. It should be mentioned that the total result may be higher than the results in Table 4. The reason is that some CVE may belong to multiple kinds of CWE. It will be counted multiple times when counting the number of CWEs.

Then we will explain Table 6. As for T_{cross}, the highest one is CWE-444, which achieves the accuracy of 60%. If only the highest number of predictions is considered, it is CWE-502, which contains 311 correct predictions. As for T_{cwe}, the highest one is CWE-306 and it achieves a surprising prediction performance of 100%. If only the highest number of predictions is considered, it is CWE-22, which contains ten correct predictions. Detailed results are given in Table 6. *CWE No.* indicates the CWE number. The first column of *Accu* is the right prediction number and the total prediction number. The second column of *Accu* is prediction accuracy. We can find that most of the TOP CWE predictions in the two test sets are the same. CWEs with large differences will be labeled. CWEs in T_{cwe} contain less CWE categories than T_{cross}, which may have contributed to the greater concentration of top CWE. In the following, we will compare the difference between these two test sets and make a detailed analysis of why the model performs well on certain specific CWEs. They perform differently or even achieve zero accuracies in one dataset. First of all, it must be stated that the reason why these CWEs marked blue are not present on the right side is that they are not included in T_{cwe}. These will not be the focus of our attention.

Case Study: CWE-306: CWE-306 means "Missing Authentication for Critical Function". It is special because it

has a very small sample but makes a correct prediction. The software does not perform any authentication for functionality requiring a provable user identity or consuming significant resources. This commit contains two code changes as shown in Figure 17. The first one (second line) is to add the annotation "@SuppressWarnings ("resource")" before the method declaration. The second one is to modify two parameters in the put method.

```
> public static JMXConnectorServer createJMXServer (int port, boolean local) throws IOException
= @SuppressWarnings ( "resource" ) public static JMXConnectorServer createJMXServer (int port, boolean local) throws IOException
< @SuppressWarnings ( "resource" ) public static JMXConnectorServer createJMXServer (int port, boolean local) throws IOException
```

```
> env.put(RMIExporter.EXPORTER_ATTRIBUTE, new Exporter())
= env.put(jmx.remote.x.daemon, true)
< env.put(jmx.remote.x.daemon, true))
```

Fig. 17: Case: right prediction of CWE-306

These two modifications have been correctly captured and predicted by SeqTrans. The other two incorrect predictions belong to variable definition changes, the model does not make the correct prediction.

Case Study: CWE-362: CWE-362 means "Concurrent Execution using Shared Resource with Improper Synchronization". The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently. It contains a list of condition operator changes and parallelism-related modifications.

```
> private boolean closed = false
= private volatile boolean closed = false
< private static boolean closed = false
```

```
> return !getSocket().isOpen()
= return closed || !getSocket().isOpen()
< return closed || !getSocket().isOpen()
```

Fig. 18: Case: wrong prediction of CWE-362

In Figure 18, developers added one keyword and changed the return condition. The condition modification of the statement has been correctly predicted by SeqTrans. However, the addition of the volatile keyword was not successfully predicted by T_{cwe}'s model. We think the reason is that T_{cross}'s model learns from other records about adding the static keyword.

Case Study: CWE-502: CWE-502 means "Deserialization of Untrusted Data". The application deserializes untrusted data without sufficiently verifying that the resulting data will be valid. CWE-502 related code transformations account for half of the entire training set. It contains large numbers of repetitive code transformations, such as deleting one throw exception, adding a return statement, and changing parameter orders. We will list some typical code changes that are well captured and handled by SeqTrans.

```
> throw data.instantiationException(_valueClass, ClassUtil.getRootCause(cause))
= return data.handleInstantiationProblem(_valueClass, root, ClassUtil getRootCause(cause))
< return data.handleInstantiationProblem(_valueClass, root, ClassUtil.getRootCause(cause))
```

Fig. 19: Case: right prediction of CWE-502

In Figure 19, developers delete the throw keyword and add a return keyword to transfer the instantiation problem. In addition, a new parameter was inserted into the second position. This code transformation can be well captured by SeqTrans.

```
> if (type.isAssignableFrom(raw))
= if (raw.getParameterCount( ) == 1)
< if (raw.getParameterCount( ) == 1)
```

Fig. 20: Case: right prediction of CWE-502

In Figure 20, developers firstly change the target of the method call. Then, replace the method call from "isAssignableFrom" to "getParameterCount". Finally, the conditional expression "== 1" is added. This code transformation contains three single code transformations but is also well captured by SeqTrans. In general, our tool SeqTrans performs stable and outstandingly for vulnerability fixes like CWE-502 that contain a lot of repetitive code transformations.

Case Study: CWE-78 and CWE-74: These two CWEs face the same problem and we will explain them together. CWE-78 means "Improper Neutralization of Special Elements used in an OS Command". The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralize special elements that could modify the intended OS command when sent to a downstream component. CWE-74 means "Improper Neutralization of Special Elements in Output Used by a Downstream Component". The software constructs all or part of a command, data structure, or record using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralize special elements that could modify how it is parsed or interpreted when it is sent to a downstream component. We give the following explanation for the 0% accuracy of these two CWEs: T_{cwe} does not contain any of them in the training set. All of them are included in the test set. We believe that this situation is the cause of the low accuracy rate.

The conclusion reached is that, for some CWEs that contain duplicate vulnerability fixes or can be learned from historical repair records, our SeqTrans performs very well. Another hypothesis is that training a general model to fix vulnerabilities automatically is too ambitious to cover all cases. If we can focus on specific types of CWEs, the NMT model can make a very promising result to help developers.

Answer to RQ3. Finding 1: SeqTrans performs well in predicting specific kinds of vulnerability fixes like CWE-287 and CWE-362. It also performs well on a timing test set that simulates learning historical modification records. The prediction range will become wider and wider as the historical repair records increases.

On the other hand, to deeply analyze these specific CWEs, we derived Table 7 that shows the classification of code transformations by manually analyzing prediction results and source codes. We have made a change type classification for each code change not only the correct prediction but also the wrong prediction. We only consider the prediction results strictly consistent with the true modifications as correct predictions. So the actual accuracy should be higher than the strict matching calculation method we used. The first column is the type name of code transformations. We roughly divided the code transformation types into 17 categories. It is worth noting that some single predictions can include multiple types of code changes, they are classified into different code change types. For this reason, the sum of the classified changes is not equaled to the number in Table 6. Detailed definitions are shown in the following:

- Change Parameter: Add, delete the parameter or change the parameter order.
- Change Throw Exception: Add, delete or replace the block of throw exception, add or delete the exception keywords in the method declaration.
- Change Variable Definition: Change variable type or value.
- Change Method Call: Add, delete a method call or replace a method call by another.
- Change Target: Maintain the same method call but change the target of the method call.
- Change String: Add, delete or replace the string.
- Change Method Declaration: Add, delete or replace method name and the qualifier.
- Change Class Declaration: Modify the declaration of a class.
- Change if Condition: Add, delete or replace operands and operators in the if condition.
- Change Switch Block: Add, delete or replace the "case" statement.
- Change Loop Condition: Modify the loop condition.
- Change Return Statement: Change return type or value, add or delete "return" keyword.
- Change Keywords "this/super": add or delete these keywords.
- Change Try Block: Put statements into the try block.
- Change Catch Exception: Add, delete or replace the block of catch exception.
- Refactoring: Rewrite the code without changing functionality.
- Other: Other transformations which are hard to be categorized or occur infrequently.

We can observe some conclusions from Table 7. In T_{cross}, SeqTrans performs well in predicting throw exception, string, and keywords changes. All of them substantially above average accuracy. When predicting parameter change, method declaration, and variable definition. SeqTrans also performs better than the average accuracy. In T_{cwe}, SeqTrans performed consistently with T_{cross}. Only class declaration, switch block, loop condition, catch exception changes, and refactoring show lower accuracy than others. We believe this gap can be explained in two points: code change sophistication and relevance. There are certain templates for code changes like string and throw exceptions. SeqTrans can more easily learn how to modify such changes from historical data. But some of code transformations involve sophisticated code changes[1], while others may only be due to insufficient samples, resulting in the model not learning well. On the other hand, code changes such as refactorings and switch structure changes are difficult to accomplish with independent statement changes because the code is so interconnected. This also leads to a decrease in model prediction accuracy.

TABLE 7: Types of code transformation learned by SeqTrans

Code Transformations	Accu	
	T_{cross}	T_{cwe}
Change Parameter	126/495(25.5%)	17/49(34.7%)
Change Throw Exception	98/227(43.1%)	5/15(33.3%)
Change Variable Definition	63/265(23.8%)	11/33(33.3%)
Change Method Call	41/194(21.1%)	4/11(36.4%)
Change Target	19/123(15.4%)	2/13(15.4%)
Change String	79/178(44.4%)	12/21(57.1%)
Change Method Declaration	47/197(23.9%)	3/13(23.1%)
Change Class Declaration	1/57(1.8%)	0/3(0%)
Change If Condition	28/167(16.8%)	2/7(28.6%)
Change Switch block	3/31(9.7%)	0/2(0%)
Change Loop Condition	2/38(5.3%)	0/2(0%)
Change Return Statement	31/180(17.2%)	4/14(28.6%)
Change Keywords "this/super"	7/18(38.3%)	1/5(20.0%)
Change Try Block	2/17(11.8%)	1/3(33.3%)
Change Catch Exception	1/13(7.7%)	0/1(0%)
Refactoring	4/85(4.7%)	0/1(0%)
Other	7/22(31.8%)	1/6(16.7%)

Answer to RQ3. Finding 2: SeqTrans performs well in handling throw exception change, string change and keywords change in both datasets. Simple code transformations is easier to be learned by the model, even in unseen situations. Sophisticated code and strongly correlated code transformations is not easily modified.

Overall, SeqTrans will perform well above average against specific kinds of CWE and specific kinds of code transformations. As the model iterates in the hands of developers and the size of the data increases, we believe SeqTrans has much space for improvement.

6 Discussion

6.1 Internal Threats

The performance of the NMT model can be significantly influenced by the hyperparameters we adopted. The transformer model is susceptible to hyperparameters. In order to mimic the Google setup, we set a bunch of options suggested by OpenNMT [64] to simulate their result. However, there are gaps between source code language and natural language. We also modified and tested part of the hyperparameters and chose the one that achieved the best performance.

1. CVE-2015-5171, UAA, 9730cd6a3bbb481ee4e400b51952b537589c469d

We manually analyzed the prediction result and the source code, classified them into 17 types. This number of categories is based on our experience during the experiment process, which may not be complete enough to cover all the code transformations. More refined classification may lead to more discoveries. However, during our analysis, we found that most code changes can be categorized into specific code transformations or a list of them. Only a few code changes cannot be identified, classified, and even partly should be attributed to the mismatch of Gumtree [53]. In addition, there is the potential to introduce human error in the validation process. We have taken our best efforts to avoid human errors. All the validators in the experiments have more than three years of experience in Java development.

The small dataset and the complex transformer model may face the overfitting problem, which is occurred for three reasons: a small dataset, too many training steps and a complex model which is not fully trained. In this work, we referenced He's work [73] and applied a pre-training model to alleviate it. He's work proposes the following observation:

1) Training from scratch is not a bad choice, either.
2) Pre-training allows the model to be converged earlier.
3) When the amount of material is small, the pre-trained model is less likely to be over-fitted.
4) Pre-training is helpless for tasks that are not very homogeneous.

We think our specific domain dataset meets the above conditions. The transformer model is more complex than the seq2seq model, which contains more parameters to be fully trained. Pre-training will speed up convergence on the target task. Applying a pre-training model will be helpful to alleviate the overfitting problem. Our experimental results have also confirmed this opinion.

6.2 External Validity

During the experiment, we find that Gumtree [53] will introduce mismatches, which will affect the quality of the training set. Other researchers have mentioned that occasionally GumTree cannot appropriately detect motion and update actions between two ASTs [74], [75]. In fact, we found two problems with Gumtree, one is related to the IO issue. We found that the IO streams Gumtree used can cause blockages, and this has been confirmed and fixed by Gumtree's author. Another problem is in the bottom-up algorithm part of Gumtree. This question did not receive a response from the author. Neither did we do further experiment to evaluate the false-positive rate. Verifying this problem is very difficult, and we have difficulty collecting a suitable ground truth. We also modified Gumtree to support statement-level code matching and def-use chain collection. We believe that through these, we have minimized the impact of Gumtree.

In addition, although we did not directly include fault localization in our evaluation of SeqTrans, we have also done some experiments related to fault location accuracy. We have investigated the popular fault localization tools and finally chose SpotBugs [76]. It contains a plugin named Find Security Bugs [77], designed to detect 138 different vulnerability types with over 820 unique API signatures. We have compared the bug reports provided by Spotbugs with our known vulnerability locations provided by the fix records. Unfortunately, SpotBugs can only detect about 15% of the vulnerability locations correctly. This result is beyond our expectations. This low result shows that vulnerability localization is such a difficult work. The latest automatic program repair tools can still only be used to assist developers. There is still a long way to separate from the developers and independently do accurate automatic program fixes. Exploring how to combine fault localization and automatic program repair together will be an important future work for us.

6.3 Limitations

The main limitation of SeqTrans is that it currently only supports the single-line prediction. We always assume that these vulnerable statements are independent of each other when making predictions about the full CVEs. We plan to abstract and tokenize the vulnerable function at the function-level, and the data format we currently use cannot handle this length quite well.

6.4 Applications

We believe SeqTrans can help programmers reduce repetitive work and give reasonable recommendations for fixing vulnerable statements. As SeqTrans receives more and more modification records from developers, we believe there is still space for improvement in its performance. We have also developed a VSCode plugin of SeqTrans to provide suggestions for developers to improve their codes, which will be opened soon.

On the other hand, training a generic model on large-scale data is very expensive, and it takes a long time to adjust the hyperparameters. It would be meaningful work to provide a general model for subsequent researchers to refine directly based on this model.

The source code of SeqTrans is available at https://github.com/chijianlei/SeqTrans.

This approach can also be applied to areas outside of vulnerability fixing, such as fine-grained code refactoring. We can use historical knowledge to refactor target code such as attribute extraction, merge parameter, inline variable, etc. This is also part of our future exploration work. Moreover, our study is based on the Java language now. However, we believe that there is a common logic between programming languages, and the rules and features learned by the model can be easily applied to other languages.

7 Related Works

In recent years, Deep Learning (DL) has become a powerful tool to solve problems of Software Engineering (SE), which can capture and discover features by the DL model rather than manual derivation. In this work, we apply the Neural Machine Translation (NMT) model into the program repair field to learn from historical vulnerability repair records, summarize common pattern rules to apply to subsequent

vulnerability fixes. In the following, we will introduce studies focus on program repair and compare our work with related research.

Automated Program Repair Traditional program repair techniques can be categorized into two main categories: heuristic-based [42], constraint-based [42]. These techniques can sometimes be enhanced by machine learning, which we call learning-based repair [42]. It should be noted that the classification between these three approaches is vague, many techniques use more than one of them simultaneously. We will list some traditional techniques to explain these three types of approaches.

Heuristic-based APR approaches construct and traverse the search space for syntax program modifiers [42]. ARJA-e [78] proposes a new evolutionary repair system for Java code that aims to address challenges for the search space. SimFix [79] utilizes both existing patches and similar code. It mines an abstract search space from existing patches and obtains a concrete search space by differencing with similar code snippets. Gatafix [80] is based on a novel hierarchical clustering algorithm that summarizes fix patterns into a hierarchy ranging from general to specific patterns. GenProg [6] and RSRepair [13] are two similar approaches. Both of them try to repair faulty programs with the same mutation operations in a search space. But GenProg uses random search, rather than genetic programming, to guide the patch generation process. Meditor [26] provides a novel algorithm that flexibly locates and groups MR (migration-related) code changes in commits. For edit application, Meditor matches a given program with inferred edits to decide which edit is applicable and produce a migrated version for developers. AppEvolve [28] can automatically perform app updates for API changes based on examples of how other developers evolved their apps for the same changes. This technique is able to update 85% of the API changes considered, but it is quite time-consuming and not scalable enough.

Some approaches mine and learn fixing patterns from prior bug fixes. SimFix [79], FixMiner [32], ssFix [81], CapGen [31] and HDRepair [82]are based on frequently occurred code change operations that are extracted from the patches in code change histories. The main difference between them is the object from which the data is extracted and how the data is processed. AVATAR [33] exploits fix patterns of static analysis violations as ingredients for patch generation. SOFix [83] has a novel approach to digging up bug fix records from Stack Overflow responses.

These studies are still based on statistical ranking or strict context matching. However, more and more studies are beginning to exploit machine learning to rank similar code transformations and automatically generate code recommendations.

Constraint-based APR approaches usually focus on fixing a conditional expression, which is more prone to defects than other types of program elements. Elixir [84] uses method call-related templates from par with local variables, fields or constants, to construct more expressive repair expressions, that go into synthesizing patches. ACS [85] focuses on fine-grained ranking criteria for condition synthesis, which combines three heuristic ranking techniques that exploit the structure of the buggy program, the document of the buggy program, and the conditional expressions in existing projects.

Learning-based APR approaches is actually part of heuristic-based APR approaches that are enhanced by machine learning techniques. We have separated them as an independent category. DeepFix [36] is a program repair tool using a multi-layered sequence-to-sequence neural network with attention for fixing common programming errors. In a collection of 6,971 incorrect C language programs written by students for 93 programming tasks, DeepFix can completely repair 1881 (27%) of them, and can partially repair 1338 (19%) of them. HERCULES [86] presents an APR technique that generalizes single-hunk repair techniques to include an important class of multi-hunk bugs, namely bugs that may require applying a substantially similar patch at a number of locations. The limitation is that it addresses only a specific class of multi-hunk repairs and the evaluation is only carried out on the Defects4J dataset. TRACER [87] is another work that is very similar to Deepfix for fixing compiler errors, and its accuracy rate exceeds that of Deepfix. Tufano et al. [37], [71] has investigated the feasibility of using NMT for learning wild code. The disadvantage of his method is that only sentences with less than 100 tokens are analyzed. In addition, this work is only limited to the type of bug that contains only one sequence within a single method.

SequenceR [38] presents a novel end-to-end approach to program repair based on sequence-to-sequence learning. It utilizes the copy mechanism to overcome the unlimited vocabulary problem. To the best of our knowledge, it achieves the best result reported on such a task. However, the abstract data structure of this method retains too much useless context. It does not use the normalization method either.

Vulnerability Repair Fixing vulnerability is critical to protect users from security compromises and prevent vendors from losing user confidence. Traditional tools such as Angelix [88], Semfix [7] and ClearView [89] heavily rely on a set of positive/negative example inputs to find a patch that makes the program behaves correctly on those examples. SENX [90] propose a different approach called "property-based" which relies on program-independent, vulnerability-specific, human-specified safety properties.

Another trending direction is the application of neural network models for vulnerability repair. Harer et al. [91] apply Generative Adversarial Network (GAN) to the problem of automated repair of software vulnerabilities. They address the environment with no labeled vulnerable examples and achieve performance close to seq2seq approaches that require labeled pairs. Chen et al. [92] apply the simple seq2seq model for vulnerability repair but the performance is not quite promising. Ratchet [93] also utilizes the NMT model to fix vulnerabilities, but it only stores single statements without any context around them. All of these functions do not consider multiple-statement, either.

Transformer and Tree Structure Another popular direction is utilizing the deep learning model or treating source code as a syntax tree to maintain richer information. TranS^3 [94] proposes a transformer-based framework to integrate code summarization with code search. Tree-based neural network such as TreeLSTM [95], [96], ASTNN [97] or TreeNet [98] are also being applied on program analysis. Shiv et al. [99] propose a method to extend transformers

to tree-structured data. This approach abstracts the sinusoidal positional encodings of the transformer, using a novel positional encoding scheme to represent node positions within trees. It achieves a 22% absolute increase in accuracy on a JavaScript to CoffeeScript [100] translation dataset. TreeCaps [101] proposes a tree-based capsule network for processing program code in an automated way that encodes syntactical code structures and captures code dependencies more accurately. CODIT [102] and DLFix [103] has begun to apply tree structure into program repair and achieve some progress.

The most similar work to us is VRepair [104]. Both of the two studies used fine-tuning to solve the small sample problem. The size of their training set is also in the same order of magnitude as ours. The main differences between VRepair and SeqTrans are the targeted languages and data structures. VRepair focuses on the C language but the target of SeqTrans is on the Java language. Also, in order to decreases the size of the output sequence, VRepair represents edit scripts at token level and the network only outputs the changed source code tokens not the whole function. However, the problem is that multiple inference results will be generated when backfilling the modified token. In our approach, we will maintain the suspicious statements and all statements that contain data dependencies with the suspicious statements. In other words, we will preserve more context around the suspicious statements but also make sequences longer. In addition, his work does not provide a runnable example or code.

Most of these techniques focus on single statement prediction. Translating multiple statements together is more challenging than translating one language to another language. Techniques for characterizing code using tree and graph structures and converting the resulting prediction trees into readable code are still in the exploratory stage. Overall, we believe that using a tree-based neural network or even combining it with a transformer structure will become our future work.

8 Conclusion

In this paper, we design the automatic vulnerability fix tool SeqTrans based on the NMT technique to learn from historical vulnerability fixes. It can provide suggestions and automatically fix the source code for developers. Fine-tuning strategy is used to overcome the small sample size problem. We conduct our study on real-world vulnerability fix records and compare our SeqTrans with three kinds of other NMT techniques. We investigated three research questions based on these collected data. Experiment results show that our technique outperforms the state-of-the-art NMT model and achieves an accuracy rate of 23.3% in statement-level prediction and 25.3% in CVE-level prediction. The SeqTrans-based approach indeed helps solve the scalability and small data set problems of existing methods on the task of vulnerability fixing. We also look deeply into the model and manually analyze the prediction result and the source code. Our observation finds that SeqTrans performs exceptionally well in specific kinds of CWEs like CWE-287 (Improper Authentication) and CWE-863 (Incorrect Authorization). The prediction range will become wider and wider as the historical repair records increases.

9 Acknowledgement

This work was supported by National Key Research and Development Program of China (2018YFB1004500), National Natural Science Foundation of China (62002280, 61632015, 61772408, U1766215, 61833015,61902306), Innovative Research Group of the National Natural Science Foundation of China (61721002), Innovation Research Team of Ministry of Education (IRT_17R86), Project of China Knowledge Centre for Engineering Science and Technology. Project of Chinese Academy of Engineering "The Online and Offline Mixed Educational ServiceSystem for 'The Belt and Road' Training in MOOC China"

References

[1] T. Britton, L. Jeng, G. Carver, and P. Cheak, "Quantify the time and cost saved using reversible debuggers," Technical report, Cambridge Judge Business School, Tech. Rep., 2012.

[2] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.

[3] A. Zeller, *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[4] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[5] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[6] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[7] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[8] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[9] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 550–554.

[10] S. R. L. Marcote and M. Monperrus, "Automatic repair of infinite loops," *arXiv preprint arXiv:1504.05078*, 2015.

[11] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, 2011, pp. 1427–1434.

[12] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[13] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[14] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020, pp. 1–17.

[15] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[16] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[18] B. Potter and G. McGraw, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[19] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[20] Z. Shen and S. Chen, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," *Security and Communication Networks*, vol. 2020, 2020.

[21] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, 2015, pp. 1–9.

[22] M. Zalewski, "American fuzzy lop," https://github.com/google/AFL/, Spt, 2010.

[23] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[24] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[25] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 229–246.

[26] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 335–346.

[27] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *ACM Sigplan Notices*, vol. 45, no. 10, pp. 302–321, 2010.

[28] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 204–215.

[29] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 47–50.

[30] M. Lamothe, W. Shang, and T.-H. Chen, "A4: Automatically assisting android api migrations using code examples," *arXiv preprint arXiv:1812.04894*, 2018.

[31] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[32] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[33] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.

[34] I. Eclipse, "Eclipse ide," *Website www. eclipse. org Last visited: July*, pp. 1–20, 2009.

[35] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[36] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[37] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "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*, 2018, pp. 832–837.

[38] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[39] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[40] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[41] "Aosp vulnerability dataset," https://source.android.com/security/bulletin, April 20, 2021.

[42] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[43] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, "Convolutional neural networks for medical image analysis: Full training or fine tuning?" *IEEE transactions on medical imaging*, vol. 35, no. 5, pp. 1299–1312, 2016.

[44] C. Casalnuovo, K. Sagae, and P. Devanbu, "Studying the difference between natural and programming language corpora," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1823–1868, 2019.

[45] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do i use the wrong definition? defuse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 160–174.

[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[47] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[48] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://www.aclweb.org/anthology/P17-4012

[49] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, no. 3. Makuhari, 2010, pp. 1045–1048.

[50] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[51] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[52] "Cs231n: Convolutional neural networks for visual recognition," https://cs231n.github.io/transfer-learning/, Spt 20, 2021.

[53] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[54] "srcml," https://www.srcml.org/, April 20, 2021.

[55] "Scitools understand," https://scitools.com/features/, Sep 20, 2019.

[56] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[57] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. Van Der Maaten, "Exploring the limits of weakly supervised pretraining," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 181–196.

[58] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[59] "Findbugs™ - find bugs in java programs," http://findbugs.sourceforge.net/, March 06, 2015.

[60] "Eclipse jgit," https://www.eclipse.org/jgit/, Accessed April 4, 2017.

[61] T. Gowda and J. May, "Finding the optimal vocabulary size for neural machine translation," *arXiv preprint arXiv:2004.02334*, 2020.

[62] B. Thompson, H. Khayrallah, A. Anastasopoulos, A. D. McCarthy, K. Duh, R. Marvin, P. McNamee, J. Gwinnup, T. An-

derson, and P. Koehn, "Freezing subnetworks to analyze domain adaptation in neural machine translation," *arXiv preprint arXiv:1809.05218*, 2018.

[63] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," *arXiv preprint arXiv:2004.10964*, 2020.

[64] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://www.aclweb.org/anthology/P17-4012

[65] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[66] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[67] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[68] V. Raychev, M. Vechev, and 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.

[69] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," *arXiv preprint arXiv:1702.01806*, 2017.

[70] "Github compare api," https://docs.github.com/en/rest/reference/repos#commits, April 20, 2021.

[71] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[72] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[73] K. He, R. Girshick, and P. Dollár, "Rethinking imagenet pre-training," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 4918–4927.

[74] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 264–274.

[75] J. Matsumoto, Y. Higo, and S. Kusumoto, "Beyond gumtree: A hybrid approach to generate edit scripts," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 550–554.

[76] "Spotbugs," https://spotbugs.github.io/index.html, Spt 20, 2021.

[77] "Find security bugs," https://find-sec-bugs.github.io/, Spt 20, 2021.

[78] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–53, 2020.

[79] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and 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.

[80] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[81] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 660–670.

[82] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[83] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 118–129.

[84] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 648–659.

[85] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[86] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[87] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: for the student programs, from the student programs," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 2018, pp. 78–87.

[88] S. Mechtaev, J. Yi, and 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.

[89] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 87–102.

[90] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 539–554.

[91] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, L. Kim *et al.*, "Learning to repair software vulnerabilities with generative adversarial networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 7933–7943.

[92] Z. Chen, S. Kommrusch, and M. Monperrus, "Using sequence-to-sequence learning for repairing c vulnerabilities," *arXiv preprint arXiv:1912.02015*, 2019.

[93] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *arXiv preprint arXiv:1812.07170*, 2018.

[94] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "Trans^ 3: A transformer-based framework for unifying code summarization and code search," *arXiv preprint arXiv:2003.03238*, 2020.

[95] M. Ahmed, M. R. Samee, and R. E. Mercer, "Improving tree-lstm with tree attention," in *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*. IEEE, 2019, pp. 247–254.

[96] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

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

[98] Z. Cheng, C. Yuan, J. Li, and H. Yang, "Treenet: Learning sentence representations with unconstrained tree structure." in *IJCAI*, 2018, pp. 4005–4011.

[99] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Advances in Neural Information Processing Systems*, 2019, pp. 12 058–12 068.

[100] T. Burnham, *Coffeescript: accelerated Javascript development*. Pragmatic Bookshelf, 2015.

[101] V. Jayasundara, N. D. Q. Bui, L. Jiang, and D. Lo, "Treecaps: Tree-structured capsule networks for program source code processing," *arXiv preprint arXiv:1910.12306*, 2019.

[102] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, 2020.

[103] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

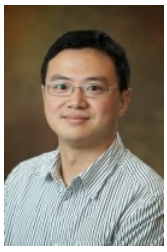
[104] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *arXiv preprint arXiv:2104.08308*, 2021.

Jianlei Chi received the B.S. degree in computer science and technology from Harbin Engineering University, China, 2014, and the Ph.D. degree in computer science and technology in 2022 from Xi'an Jiaotong University, China. He is a post-doctoral researcher at the Institute of Cyberspace Security, Zhejiang University of Technology, China. His research interests include trustworthy software, software engineering, program analysis and machine learning.

Yu Qu received the B.S. and Ph.D. degrees from Xi'an Jiaotong University, Xi'an, China in 2006 and 2015 respectively. He is a post-doctoral researcher at the Department of Computer Science and Engineering, UC Riverside. His research interests include trustworthy software and applying complex network and data mining theories to analyzing software systems.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. Currently, he is a professor of the Systems Engineering Institute, Xi'an Jiaotong University. His research interests include smart grid, network security and trustworthy software.

Qinghua Zheng Qinghua Zheng received the B.S. degree incomputer software in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xi'an Jiaotong University, China. He was a postdoctoral researcher at Harvard University in 2002. He is currently a professor in Xi'an Jiaotong University, and the dean of the Department of Computer Science. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software.



Heng Yin is a professor in the department of Computer Science and Engineering at UC Riverside. Before joining UC Riverside, he was with Syracuse University from September 2009 to June 2016, as assistant professor and then associate professor. He obtained his Ph.D in Computer Science from the College of William and Mary in 2009, while He spent 4 years at Carnegie Mellon University and later at UC Berkeley. His research interests lie in computer security and developing all kinds of techniques (such as program analysis, virtualization, and machine learning/deep learning) to solve computer and software security problems, including but not limited to malware detection and analysis, vulnerability discovery, program hardening, digital forensics.