VulRepair: A T5-Based Automated Software Vulnerability Repair

Michael Fu
yeh.fu@monash.edu
Monash University
Australia

Chakkrit Tantithamthavorn*
chakkrit@monash.edu
Monash University
Australia

Trung Le
trunglm@monash.edu
Monash University
Australia

Van Nguyen
khacvan.nguyen@adelaide.edu.au
The University of Adelaide
Australia

Dinh Phung
dinh.phung@monash.edu
Monash University
Australia

ABSTRACT

As software vulnerabilities grow in volume and complexity, researchers proposed various Artificial Intelligence (AI)-based approaches to help under-resourced security analysts to find, detect, and localize vulnerabilities. However, security analysts still have to spend a huge amount of effort to manually fix or repair such vulnerable functions. Recent work proposed an NMT-based Automated Vulnerability Repair, but it is still far from perfect due to various limitations. In this paper, we propose VulRepair, a T5-based automated software vulnerability repair approach that leverages the pre-training and BPE components to address various technical limitations of prior work. Through an extensive experiment with over 8,482 vulnerability fixes from real-world software projects, we find that our VulRepair achieves a Perfect Prediction of 44%, which is 13%-21% more accurate than competitive baseline approaches. These results lead us to conclude that our VulRepair is considerably more accurate than two baseline approaches, highlighting the substantial advancement of NMT-based Automated Vulnerability Repairs. Our additional investigation also shows that our VulRepair can accurately repair as many as 745 out of 1,706 real-world well-known vulnerabilities (e.g., Use After Free, Improper Input Validation, OS Command Injection), demonstrating the practicality and significance of our VulRepair for generating vulnerability repairs, helping under-resourced security analysts on fixing vulnerabilities.

CCS CONCEPTS

• **Software and its engineering**; • **Security and privacy**;

KEYWORDS

Software Vulnerability Repair

ACM Reference Format:
Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549098

1 INTRODUCTION

Software vulnerabilities are security flaws, glitches, or weaknesses found in software systems that could be exploited by attackers to undertake malicious activities [6]. In particular, criminal groups may make use of unresolved security vulnerabilities in software to attack and damage a system to steal confidential information or extort assets, resulting in severe economic damage [17]. According to the statistics of the National Vulnerability Database (NVD), the number of vulnerabilities discovered per year is considerably increased five times from 4k+/year in 2011 to 20k+/year in 2021 [1].

Recently, researchers proposed various Artificial Intelligence (AI)-based approaches to help under-resourced security analysts better understand the characteristics of vulnerabilities (e.g., vulnerability analysis) [2, 3, 5, 7, 44, 57, 58, 62] and find vulnerabilities faster (e.g., vulnerability predictions) [12, 20, 21, 25, 26, 34, 37, 43, 49–51, 55, 63, 65]. For example, AI-based vulnerability prediction approaches are proposed to predict if a given function is vulnerable or not at the various granularity levels (e.g., function [12, 37, 43, 65], line [20, 21, 28, 36, 38–40, 48, 59]) using various types of information (e.g., graph properties [12, 21, 55, 65], semantic [20, 43], syntactic information [12, 21, 55, 65], and mutual information [36]). Such vulnerability prediction approaches only help security analysts to find, detect, and localize the location of vulnerabilities. However, security analysts still have to spend a huge amount of effort to manually fix or repair vulnerabilities [4, 11, 35].

Recently, Chen *et al.* [13] proposed VRepair [13], an automated vulnerability repair approach that leverages a Transformer-based Neural Machine Translation (NMT). VRepair is proposed to address various challenges of prior work in the area of automated program repairs (e.g., SequenceR [14], an RNN-based NMT model). However, VRepair is still inaccurate due to the following three limitations.

Limitation ①: VRepair is trained on a small bug-fix corpus of 23,607 C/C++ functions, which may generate suboptimal vector representations.

Limitation ②: VRepair leverages the word-level tokenization and the copy mechanism to handle Out-Of-Vocabulary (OOV), limiting its ability to generate new tokens that never appear in a vulnerable function but are newly introduced in the vulnerability repair.

[*] The corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549098

Limitation ③: VRepair leverages a Vanilla Transformer (i.e., a basic Encoder-Decoder Transformer architecture) which uses an absolute positional encoding, limiting the ability of its self-attention mechanism to learn the relative position information of code tokens within the input sequences.

In this paper, we propose VulRepair, a T5-based Vulnerability Repair approach, aiming to address the aforementioned limitations of VRepair [13]. First, our VulRepair employs a pre-training CodeT5 component from a large codebase (i.e., CodeSearchNet+C/C# [23, 56] with 8.35 million functions from 8 different Programming Languages) to generate more meaningful vector representation, employs BPE tokenization to handle Out-Of-Vocabulary (OOV) issues, and employs a T5 architecture that considers the relative position information in the self-attention mechanism. Through an extensive evaluation of our VulRepair on CVEFixes [10] and Big-Vul dataset [18] consisting of 8,482 vulnerability fixes from 1,754 large-scale open-source software projects with over 180+ different CWE types spanning from 1999 to 2021, we address the following four research questions:

(RQ1) What is the accuracy of our VulRepair for generating software vulnerability repairs?
Results. Our VulRepair achieves a Perfect Prediction of 44%, which is 21% more accurate than VRepair [13] and 13% more accurate than CodeBERT [19].

(RQ2) What is the benefit of using a pre-training component for vulnerability repairs?
Results. Regardless of the model architectures, the PL/NL pre-training corpus improves the percentage of perfect predictions by 30%-38% for vulnerability repair approaches, highlighting the substantial benefits of using the pre-training component for vulnerability repair approaches.

(RQ3) What is the benefit of using BPE tokenization for vulnerability repairs?
Results. Regardless of the model architectures, the BPE subword tokenization improves the percentage of perfect predictions by 9%-14% for vulnerability repair approaches, highlighting the substantial benefits of using BPE tokenization for vulnerability repair approaches.

(RQ4) What are the contributions of the components of our VulRepair?
Results. The pre-training component of our VulRepair is the most important component. Without a proper design of T5 architecture for our VulRepair, the performance can be decreased from 44% to 1%. This finding highlights that designing an NMT-based automated vulnerability repair approach is still a challenging task, which requires a deep understanding of modern Transformer architectures to achieve the highest possible %perfect predictions.

These results lead us to conclude that our VulRepair is considerably more accurate baseline approaches, highlighting the substantial advancement of NMT-based automated vulnerability repairs. Thus, our VulRepair is expected to recommend vulnerability repair candidates to security analysts, which could reduce their limited effort on fixing vulnerable functions. Our additional investigation also shows that our VulRepair can accurately repair many real-world vulnerabilities (e.g., Use After Free, Integer Overflow, NULL Pointer Dereference). Our additional analysis discovers findings that lead to many open research challenges for future studies.

The Novelty & Contributions of this paper are as follows:

- VulRepair, a T5-based Vulnerability Repair approach, aiming to address various limitations of VRepair [13].
- An extensive evaluation of our VulRepair with two competitive baseline approaches (i.e., VRepair [13], CodeBERT [19]).
- An empirical evaluation of the impact of the pre-training component for software vulnerability repairs.
- An empirical evaluation of the impact of tokenization techniques for software vulnerability repairs.
- An ablation study to investigate the contribution of each component of our VulRepair approach.

Paper Organization. Section 2 describes the problem definition and the limitations of prior work. Section 3 presents our VulRepair approach. Section 4 presents the experimental setup, while Section 5 presents the results. Section 6 presents additional discussion. Section 7 presents the related works. Section 8 discloses the threats to validity. Section 9 draws the conclusions.

2 BACKGROUND & PROBLEM MOTIVATION

Automated Vulnerability Repair (AVR) can be formulated as a Neural Machine Translation (NMT) task [61]. Formally speaking, the objective of an NMT-based AVR model aims to learn the mapping between a vulnerable function X_i and a vulnerability repair Y_i (i.e., the repair version of X_i). In particular, a NMT-based model is composed of Encoder layers and Decoder layers, where the Encoder takes a sequence of code tokens as input to map a vulnerable function $X_i = [x_1, ..., x_n]$ into a fixed-length intermediate hidden state $H = [h_1, ..., h_n]$. Then, the decoder takes the hidden state vector H as an input to generate the output sequence of tokens $Y_i = [y_1, ..., y_m]$. We note that n (i.e., the length of the input sequence) and m (i.e., the length of the output sequence) can be different. To optimize the mapping, the parameters of the NMT-based model are updated using the training dataset with the following equation to maximize the conditional probability:

$$p(Y_i \mid X_i) = p(y_1, ..., y_m \mid x_1, ..., x_n) = \prod_{i=1}^{m} p(y_i \mid H, y_1, ..., y_{i-1})$$

Previously, Recurrent Neural Networks (RNNs) [47] are widely used as a NMT model for various SE tasks, e.g., RNN-based automated program repair approaches in SequenceR [14] and Tufano *et al.* [53]. As the length of source code grows, RNN-based models suffer from long-term dependencies among the input tokens, making the RNN-based models forget some past information for a long sequence of tokens (which is common in source code).

A Transformer-based NMT model, introduced by Google Brain [54], is an Encoder-Decoder architecture with the self-attention mechanism. Unlike RNNs, Transformers do not necessarily process the sequence of tokens in a sequential order. Instead, the self-attention mechanism provides a context vector for any position in the input sequence (i.e., the context vector is used to provide a weight of the tokens that the model should pay attention to), allowing the Transformer-based NMT architecture to be more accurate than RNN-based NMT architecture. Therefore, Chen *et al.* [13] proposed VRepair [13] (accepted at IEEE TSE 2022), which is a

Transformer-based NMT approach for automated vulnerability repair, aiming to address various limitations of RNN-based NMT approaches for automated program repairs [14, 53]. Specifically, VRepair consists of the following three steps:

Step 1: Code Representation. For an input vulnerable function, VRepair first leverages a word-level Clang tokenizer with a copy mechanism to tokenize a C function into a sequence of tokens. Then, a word embedding layer is used to generate a vector representation of each token in the sequence to capture the semantic information among the input tokens. Then, an absolute positional encoding layer is used to generate another vector representation of the same sequence that considers positional information among the input tokens. Finally, the two vector representations are added together to form the final code representation which will be used as the input vector of the encoder-decoder model.

Step 2: An Encoder-Decoder Transformer. Given the input vectors, VRepair leverages an encoder-decoder Transformer model [54] to generate vulnerability repairs. The representation first goes through a 6-layer Transformer encoder. Then, the output vector of the last Transformer encoder is fed to each of the six Transformer decoders. The output vector of the last decoder then goes through a linear layer with a softmax activation to obtain the final probability distribution of vocabulary used for repair generation.

Step 3: Beam Search for Repair Generation. Given the output vector of the Transformer decoder, VRepair then uses the beam search algorithm to generate 50 vulnerable repair candidates. However, there exist three major technical limitations.

Limitation ①: VRepair is trained on a small bug-fix corpus of 23,607 C/C++ functions, which may generate suboptimal vector representations. The quality of vector representation heavily relies on the language models of code being used. For the VRepair approach, Chen *et al.* [13] leverage a transfer learning technique in which VRepair is first pre-trained on a labeled bug-fix corpus to generate a vector representation, which is fed into a Transformer model. Then, the Transformer model is fine-tuned on the vulnerability dataset to perform vulnerability repairs. However, their pre-training data contains a limited number of C/C++ functions. Thus, the pre-trained model of VRepair may not generate the most meaningful vector representation of the source code.

Limitation ②: VRepair leverages the word-level tokenization and the copy mechanism to handle Out-Of-Vocabulary (OOV), limiting its ability to generate new tokens never appear in a vulnerable function but newly introduced in the vulnerability repair. Hindle *et al.* [22] found that source code is far more natural and repetitive than natural languages. Unlike in natural language, software developers are free to create any tokens and can make them arbitrarily complex [27], leading to excessively large vocabulary size. Karampatsis *et al.* [27] raise concerns that statistical language models of source code often suffer from large vocabularies and Out-Of-Vocabulary (OOV) issues, which could severely affect the performance of the neural language models of source code. Therefore, VRepair overcomes the OOV problem by using word-level tokenization with a copy mechanism [64]. The copy mechanism aims to directly copy/reuse a rare token from the input sequence to the output sequence [64]. However, the word-level tokenization and the copy mechanism cannot reuse tokens that never appear in the vulnerable functions to the vulnerability repairs, limiting its ability to generate new code tokens.

Limitation ③: VRepair leverages a Vanilla Transformer (i.e., a basic Encoder-Decoder Transformer architecture) which uses an absolute positional encoding, limiting the ability of its self-attention mechanism to learn the relative position information of code tokens within the input sequences. In Step ①, VRepair leverages an absolute positional encoding layer to capture the position information (i.e., the position ID) of tokens used by the Transformer model in Step ②. This means that VRepair (the vanilla version of Transformer) requires an additional representation of absolute positions (i.e., the position ID in a sequence) to its input tokens to be added to the VRepair model. However, such absolute position information is not efficiently used in the self-attention mechanism [46], limiting the ability of its self-attention mechanism to be fully aware of the position of each token in a sequence. Such limitation could make the VRepair approach pay attention to incorrect code tokens (e.g., parenthesis instead of variable names), leading to inaccurate generation of vulnerability repairs.

3 VULREPAIR: A T5-BASED VULNERABILITY REPAIR APPROACH

In this section, we present our VulRepair architecture, which is a T5-based automated vulnerability repair approach.

Overview. Given a vulnerable function, in step ①, we perform subword tokenization using a Byte-Pairs Encoding (BPE) approach based on a CodeT5 pre-trained language model [56] to produce subword-tokenized functions (i.e., a list of subword code tokens for each function). In Step ②, we build a VulRepair model based on a T5 architecture [42]. For each subword-tokenized function, in Step (2a), VulRepair performs a word embedding to generate an embedding vector for each token and combine it into a matrix. Then, in Step (2b), the matrix is fed into the T5 encoder stack and the output of the last T5 encoder is fed into each T5 decoder in Step (2c). In Step (2d), the output of the T5 decoder stack is fed into a linear layer with softmax activation to generate the probability distribution of the vocabulary. Finally, in Step ③, we leverage beam search on top of the probability distribution of the vocabulary to generate the final candidates as a prediction. Below, we describe the details of each step.

3.1 Code Representation

The code representation of each localized vulnerable function in our studied dataset consist of two main steps:

① **BPE Subword Tokenization.** In step ①, we leverage the Byte Pair Encoding (BPE) approach [45] to perform subword-level tokenization, which consists of two main steps. (1a) generating merge operations to determine how a word should be split, and (1b) applying merge operations based on the subword vocabularies. Specifically, BPE will split all code tokens into sequences of characters and identify the most frequent symbol pair (e.g., the pair of two consecutive characters) that should be merged into a new symbol. BPE is an algorithm that will split rare tokens into meaningful subwords and preserve the common tokens (i.e., will not split the common words into smaller subwords) at the same time. For instance, the function name, $IsValidSize$, will be split into

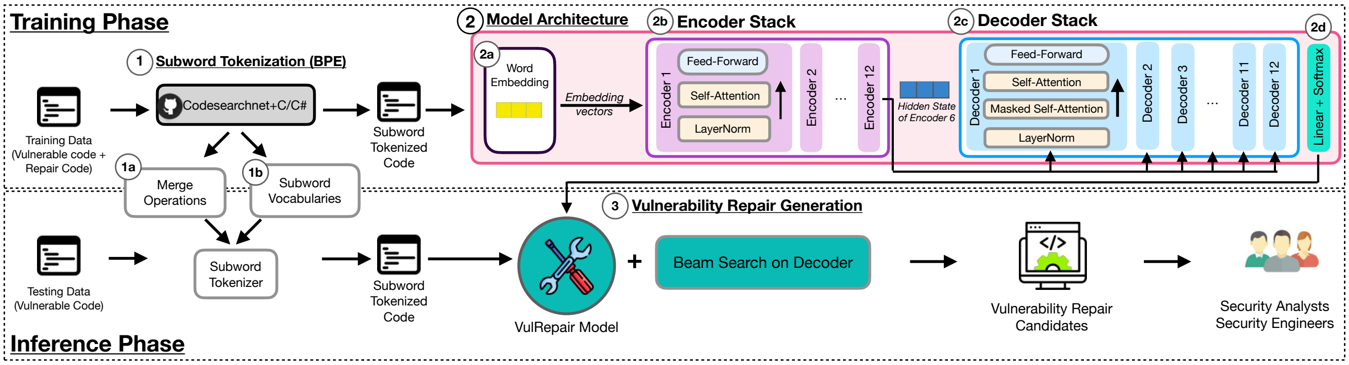

Figure 1: An overview architecture of our VulRepair.

a list of subwords, i.e., ["IsValid", "Size"]. The rare word *IsValidSize* is split into two common words, *IsValid* and *Size*.

The use of BPE subword tokenization will help reduce the vocabulary size when tokenizing various tokens because it will split rare tokens into multiple subwords instead of adding the full tokens into the vocabulary directly. In this paper, we apply a BPE tokenizer that is pre-trained on CodeSearchNet (CSN) [23] and a C/C# corpus extracted by Wang *et al.* [56]. The tokenizer is pre-trained in eight different programming languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#), which is suitable for tokenizing source code. To fit the code generation task, we add "<s>" and "</s>" tokens to represent the beginning of a sequence (BOS) and the end of a sequence (EOS). The "<pad>" token is used to pad the input sequence into the same length if needed. In addition, we add four special tokens (i.e., "<StartLoc>", "<EndLoc>", "<ModStart>", "<ModEnd>") into the vocabulary as an extra vocab ID, so they will not be split into subcomponents during tokenization. The use of pre-trained language models of source code will ensure that the vector representation being generated is more meaningful than VRepair, since it is pre-trained on a larger code corpus (i.e., CodeSearchNet+C/C#). In addition, the use of BPE will ensure that new identifiers that never appear in the vulnerable functions can be generated in the vulnerability repairs [52].

(2a) **Word Embedding.** Source code consists of multiple tokens where the meaning of each token heavily relies on the context (i.e., surrounding tokens) and the position of each token in a function. Therefore, it is important to capture the code context and its position within the function. The purpose of this step is to generate embedding vectors that capture the semantic meaning of code tokens and their position within a function. For each sub-word tokenized function, in Step (2a), we generate an [1x768] embedding vector for each subword token and combine it into a matrix to represent the meaningful relationship between a given code token and the other code tokens. To capture the semantic meaning of code tokens, we leverage word embedding vectors that are pre-trained on the same corpus as our pre-trained tokenizer discussed above. To capture the position of each code token within the function, our VulRepair leverages relative position embedding which will be computed and added to key matrix and value matrix during self-attention calculation.

3.2 VulRepair Model Architecture

VulRepair is a T5-based model [42] which starts with an encoder stack and a decoder stack, and ends with a linear layer with softmax activation.

(2b) **An Encoder Stack.** In Step (2b), a stack of twelve layers of encoder blocks is implemented to derive the encoder hidden state used by the decoders. Similar to original Transformer Encoder [54], each encoder block starts with a Layer Normalization [9] where the activation is only rescaled and no additive bias is applied [42]. Each encoder block consists of two subcomponents: a multi-head self-attention layer with relative position encoding [46] followed by a feed-forward neural network. Each subcomponent (i.e., self-attention and FFNN) in each encoder has a residual connection around it and it is followed by a layer normalization step [54].

The self-attention mechanism [54] computes the relevant scores of each code token using the dot product operation where each token interacts with itself and other tokens once. The self-attention mechanism relies on three main vectors, Query, Key, and Value. The Query is a representation of the current code token used to score against all the other tokens based on their keys stored in the Key vectors. The attention scores of each token are obtained by taking the dot product between all of the Query vectors and Key vectors. The attention scores are then normalized to probabilities using the Softmax function to get the attention weights. Finally, the Value vectors can be updated by taking the dot product between the Value vectors and the attention weight vectors.

Different from VRepair that leverages an absolute positional encoding layer with a word embedding layer to capture the positional information in the input sequence, we use a relative positional encoding to efficiently consider the representations of the relative positions and the distances between tokens within the input sequence (i.e., the relation-aware self-attention mechanism). The self-attention used in our VulRepair is a scaled dot-product self-attention with relative position encoding. The self-attention operation is computed using four matrices, i.e., Q, K, V, and P. The relative positional information, P, is supplied to the model as an additional component to the Key matrix and Value matrix as follows: $Attention(Q, K, V) = \text{softmax}(\frac{Q(K+P)^T}{\sqrt{d_k}})(V + P)$, where P is an edge representation for the two inputs in dot-product operation to determine the positional information between tokens. Different from absolute positional encoding that leverages a fixed embedding for each position, the pairwise positional encoding produces a different learned embedding according to the offset between the K and Q in the self-attention operation. Therefore, it can effectively capture the relative information among tokens.

To capture richer semantic meanings of the input sequence, we use a multi-head mechanism to realize self-attention, which allows the model to jointly attend to the information from different code representation subspaces at different positions. For d-dimension Q, K, and V, we split those vectors into h heads where each head has $\frac{d}{h}$-dimension. After all of the self-attention operation, each head will then be concatenated back again to feed into a fully-connected feed-forward neural network including two linear transformations with a ReLU activation in between. The multi-head mechanism can be summarized by the following equation: $MultiHead(Q, K, V) = Concat(\text{head}_1, ..., \text{head}_h)W^O$, where $head_i$ = Attention(QW_i^Q, KW_i^K, VW_i^V) and W^O is used to linearly project to the expected dimension after concatenation.

(2c) **A Decoder Stack.** In Step (2c), a stack of twelve layers of decoder blocks is implemented to generate the vulnerability repairs based on the hidden states provided by the last encoder block. Similar to original Transformer Decoder [54], each decoder block starts with a Layer Normalization as in the encoder block. Each decoder block consists of three subcomponents: a masked multi-head self-attention layer with relative position encoding, a multi-head encoder-decoder self-attention with relative position encoding, and a feed-forward neural network. Same as the encoder block, each subcomponent in each decoder has a residual connection around it and it is followed by a layer normalization step. The masked self-attention is used during the training phase of our generation models to restrict the model to predict the next token without attending to the later context. Thus, the model can only attend to previous tokens during the generation. This follows the situation of the inference phase where the model will not have any later context and can only attend to previous tokens during the generation.

(2d) **Linear and Softmax Layer.** The Linear layer is a fully connected neural network that projects the vector produced by the decoder stack into a larger logits vector with the number of cells equals to the number of unique token in the vocabulary. Then, the following Softmax layer transforms the value into a probability distribution that adds up to one, which would be used to generate the final output in Step ③.

3.3 Vulnerability Repair Generation

Given the output of softmax probabilities, in Step ③, we leverage beam search to select multiple vulnerability repair candidates for an input sequence at each timestep based on a conditional probability. The number of repair candidates relies on a parameter setting called Beam Width β. Specifically, the beam search selects the best β repair candidates with the highest probability using a best-first search strategy at each timestep. The beam search will be terminated when the EOS token (i.e., "</s>") is generated.

4 EXPERIMENTAL DESIGN

In this section, we present the motivation for our four research questions, our studied dataset, and our experimental setup.

Table 1: Descriptive statistics of the studied dataset.

	1st Qt.	Median	3rd Qt.	Avg.
#Tokens in Vul. Func.	138	280	593	586
#Repaired Tokens	12	24	48	55
CC. of Vul. Func.	3	8	19	23

CC: Cyclomatic Complexity

4.1 Research Questions

The key goal of this paper is to evaluate our VulRepair and compare it with two baseline approaches. Below, we present the motivation for the following four research questions.

(RQ1) What is the accuracy of our VulRepair for generating software vulnerability repairs? Recently, Chen *et al.* [13] proposed VRepair, an NMT-based automated vulnerability repair approach. However, as pointed out in Section 2, VRepair has three key limitations, leading to inaccurate vulnerability repair generation. To address these challenges, we propose our VulRepair approach. Thus, we formulate this RQ to investigate the accuracy of VulRepair when comparing to two competitive baseline approaches, i.e., VRepair (a Vanilla Transformer) [13] and CodeBERT (developed by Microsoft Research) [19, 33].

(RQ2) What is the benefit of using a pre-training component for vulnerability repairs? The quality of vector representation heavily relies on the language models of code being used. As pointed out in Section 2, VRepair is trained on a bug-fix corpus of 23,607 C/C++ functions. However, such a limited amount of data could lead to a suboptimal vector representation of code (i.e., Limitation ①, pre-training). In contrast, our VulRepair leverages a pre-trained language model of code that is pre-trained on CodeSearchNet+C/C# [23, 56] with 8.35 million functions from 8 different Programming Languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#). Thus, we formulate this RQ to investigate the impact of the pre-training component on the accuracy of automated vulnerability repair approaches.

(RQ3) What is the benefit of using BPE tokenization for vulnerability repairs? As pointed out in Section 2, VRepair leverages word-level tokenization with a copy mechanism. Such an approach may not be able to handle vulnerability repairs that have newly introduced tokens (i.e., Limitation ②, OOV problems). Recently, Karampatsis *et al.* [27] raise concerns that different tokenization approaches may have an impact on the accuracy of the language models of source code. However, the impact of tokenization approaches has not been investigated in the context of automated vulnerability repairs. Aligning with Karampatsis *et al.* [27], we formulate this RQ to investigate the impact of the tokenization component on the accuracy of vulnerability repair approaches.

(RQ4) What are the contributions of the components of our VulRepair? Our VulRepair involves various key components (i.e,. BPE+Pre-Training+T5). However, little is known about what are the contributions of the components of our VulRepair and which component contributes the most to the accuracy of our VulRepair. Thus, we formulate this RQ to conduct an ablation study on the variants of our VulRepair, where each component is altered to others while having the same T5 architecture.

4.2 Studied Dataset

In our experiment, we use the vulnerability repairs dataset, CVE-Fixes [10] and Big-Vul [18], that contains 8,482 vulnerability fixes (a pair of vulnerable C functions and vulnerable repairs). Table 1 presents the descriptive statistics of the experimental dataset.

To ensure a fair comparison with VRepair, we strictly follow the replication package provided by Chen *et al.* [13] to pre-process the experimental dataset. Each input sequence contains a special tag that specifies the CWE type of the sequence. Each vulnerable code snippet in the input sequences is labeled using the special tags "<StartLoc>" and "<EndLoc>", where "<StartLoc> indicates the beginning of the vulnerable code snippet, which will be ending with the special tag "<EndLoc>". For the output labels, each repair code snippet is represented as the special tags "<ModStart>" and "<ModEnd>", where "<ModStart> indicates the beginning of the vulnerable repair and non-vulnerable context, which will be ending with the special tag "<ModEnd>". The main purpose of adding such special tags to the tokenizer is to ensure that such special tags will not be treated as regular code tokens and will not be split by the tokenizer. Similarly, such special tags will help the model to pay attention to the areas of vulnerable code snippets and the vulnerability repair.

4.3 Experimental Setup

Split. Same as Chen *et al.* [13], we split the experimental dataset into 70% of training, 10% of validation, and 20% of testing data.

Model Implementation of Vulnerability Repair. We build our VulRepair approach on top of two deep-learning Python libraries, i.e., Transformers [60] and PyTorch [16]. The Transformers library provides API access to the transformer-based model architectures and the pre-trained weight, while the PyTorch library supports the computation during the training process (e.g., back-propagation and parameter optimization).

Model Training of our VulRepair. We obtain the CodeT5 tokenizer and model pre-trained by Wang *et al.* [56] from the API of the Transformers library. We use our training dataset to fine-tune the pre-trained model to get suitable weights for our vulnerability repair task. The model is fine-tuned on an NVIDIA RTX 3090 graphic card and the training time is around 5 hours.

In the training process, we use the cross-entropy loss ($H(p,q) = -\sum_{x \in \mathcal{X}} p(x)\ log\, q(x)$) to update the model and to optimize between each position in the predicted sequence and each position in the ground-truth sequence where $\mathcal{X}$ is the set of classes (i.e., X is a set of possible tokens generated by the BPE tokenizer for our approach and any NMT-based models like VRepair), p is the ground truth probability distribution, and q is the predicted probability distribution. To obtain the best-fine-tuned weights, we use the validation set to monitor the training process by epoch, and the best model is selected based on the optimal loss value against the validation set (not the testing set).

Hyper-Parameter Settings for Fine-Tuning. For the model architecture of our VulRepair approach, we use the default setting of CodeT5 [56], i.e., 12 Transformer Encoder blocks, 12 Transformer Decoder blocks, 768 hidden sizes, and 12 attention heads. During fine-tuning, the learning rate is set to 2e^{-5} with a linear schedule where the learning rate decays linearly throughout the training process. We use backpropagation with AdamW optimizer [31] which is widely adopted to fine-tune Transformer-based models to update the model and minimize the loss function.

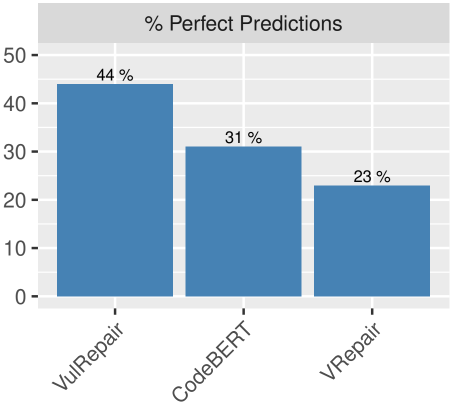

Figure 2: (RQ1) The experimental results of our VulRepair and the two baseline comparisons for vulnerability repairs. (↗) Higher % Perfect Predictions = Better.

5 EXPERIMENTAL RESULTS

(RQ1) What is the accuracy of our VulRepair for generating software vulnerability repairs?

Approach. To answer this RQ, we evaluate the accuracy of vulnerability repair approaches using the percentage of perfect predictions (%Perfect Predictions). The %Perfect Predictions measures the percentage of vulnerable functions that an approach can correctly generate a vulnerable repair that is exactly matched with ground-truth data (i.e., a human-written vulnerable repair). Then, we compare %Perfect Predictions of our VulRepair with the two baseline approaches as follows:

(1) VRepair [13] uses a vanilla Encoder-Decoder Transformer model [54] for vulnerability repairs. VRepair is first trained on a labeled bug-fixing dataset and fine-tuned on a vulnerability dataset to generate vulnerability repairs;
(2) CodeBERT [19] is an Encoder-only Transformer-based model that is pre-trained on a large codebase called CodeSearchNet [23], developed by Microsoft Research. CodeBERT consists of twelve Transformer Encoder Blocks plus six layers of Transformer Decoder for generation tasks. Mashhadi and Hemmati [33] leverage CodeBERT for automated program repair of Java bugs and present substantial improvement over RNN-based models.

For each approach, we use the same experimental setup as Chen *et al.* [13] to evaluate the accuracy of our approach and the baseline approaches. Specifically, we leverage a beam width of 50 to generate 50 repair candidates for each vulnerable function in the testing dataset. Therefore, the %Perfect Predictions can be computed as the total number of correct predictions divided by the total number of functions in the testing dataset.

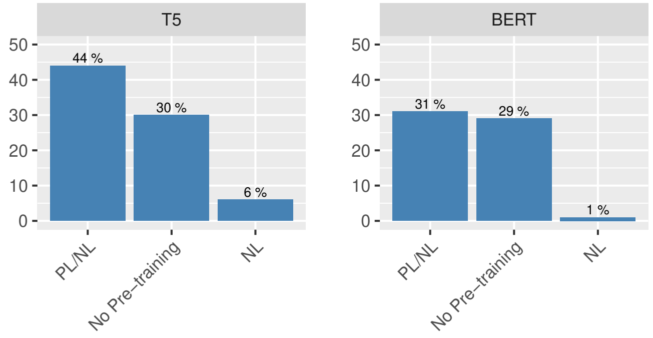

Figure 3: (RQ2) The experimental results of the ablation study with six different models. (↗) Higher % Perfect Predictions = Better.

Result. Figure 2 presents the experimental results of our VulRepair and the two baseline approaches according to our evaluation measures (i.e., %Perfect Predictions).

Our VulRepair achieves a Perfect Prediction of 44%, which is 13%-21% more accurate than the baseline approaches. Figure 2 shows that CodeBERT achieves a Perfect Prediction of 31%, while VRepair achieves a Perfect Prediction of 23%, indicating that VulRepair is 13% (44% − 31%) and 21% (44% − 23%) better than CodeBERT and VRepair respectively. The 91% accuracy improvement over the VRepair [13] has to do with different improvements of our VulRepair to address various limitations of VRepair [13], i.e., using a pre-training model from a larger codebase (i.e., CodeSearchNet+C/C# [23, 56] with 8.35 million functions from 8 different Programming Languages) to generate more meaningful vector representation (Limitation ①), using BPE tokenization to handle Out-Of-Vocabulary (OOV) issues (Limitation ②), and using T5 architecture that considers the relative position information in the self-attention mechanism (Limitation ③). In contrast, VRepair [13] leverages a word-level tokenization with a pre-trained model on a bug-fix corpus of 23,607 C/C++ functions and a vanilla Transformer (i.e., a default Transformer architecture).

(RQ2) What is the benefit of using a pre-training component for vulnerability repairs?

Approach. To answer this RQ, we aim to investigate the impact of the pre-training corpus component for vulnerability repairs. We note that VRepair leverages a vanilla Transformer, which performs model training in supervised learning (i.e., regular model training with label datasets). Thus, the model training process of VRepair is different from modern Transformer architectures like BERT and T5. In contrast, our VulRepair leverages T5 which performs a pre-training in unsupervised learning (i.e., labels are not required), i.e., the process of training a model for a general task (e.g., next word prediction) with a very large dataset. Thus, we only conduct an experiment of different pre-training corpus components with the T5 and BERT architectures. Specifically, we extend our experiment to systematically evaluate the following six variants of DL-based vulnerability repair approaches, i.e., 3 pre-training corpora (PL/NL, NL, No Pre-training) × 2 model architectures (T5, BERT).

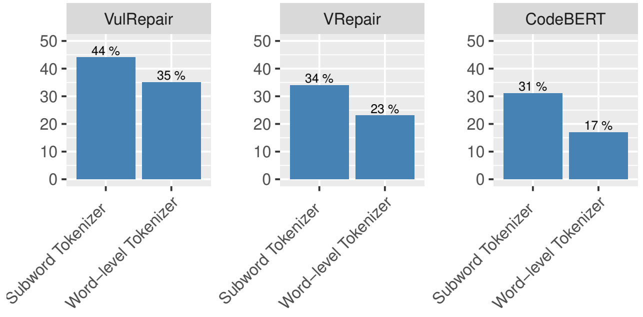

Figure 4: (RQ3) The experimental results of various approaches with different tokenization techniques for vulnerability repairs. (↗) Higher %Perfect Predictions = Better.

- ***T5 + PL/NL (our VulRepair):*** A T5 architecture that is pre-trained on both Programming Languages and Natural Language (PL/NL).
- ***T5 + NL:*** The original T5 architecture that is pre-trained on NL only (e.g., Internet webpages).
- ***T5:*** A T5 architecture without pre-training.
- ***BERT + PL/NL (original CodeBERT):*** A BERT architecture that is pre-trained on PL/NL.
- ***BERT + NL:*** The original BERT architecture that is pre-trained on NL only (e.g., Internet webpages).
- ***BERT:*** A BERT architecture without pre-training.

Similarly, we evaluate the accuracy of these variants using the same evaluation measure (i.e., % Perfect Predictions).

Result. Figure 3 presents the experimental results of the benefits of using a large pre-training corpus for vulnerability repairs.

Regardless of the model architectures, the PL/NL-based pre-training corpus improves the percentage of perfect predictions by 30%-38% for vulnerability repair approaches. Figure 3 shows that the PL/NL pre-training corpus improves %Perfect Predictions by 30% (31% − 1%) for the BERT architecture and 38% (44% − 6%) for the T5 architecture when they are trained on the NL-only corpus (i.e., the original T5/BERT architecture provided by the Transformer library). This finding highlights the substantial benefits of the pre-training process on the larger codebase, i.e., CodeSearchNet+C/C# [23, 56] with 8.35 million functions from 8 different Programming Languages (i.e., Ruby, JavaScript, Go, Python, Java, PHP, C, C#) for vulnerability repairs. Nevertheless, when using the same PL/NL pre-training corpus, **the T5 architecture employed by our VulRepair still outperforms the BERT architecture.** Figure 3 shows that, when using the same PL/NL pre-training corpus, the T5 architecture outperforms the BERT architecture by 13% (44% − 31%), highlighting the substantial benefits of the Text-to-Text Encoder-Decoder Transformer (T5) for vulnerability repairs (i.e., code→code generation) over the Encoder-only Transformer (like BERT), which could be more suitable for other SE tasks (code→text) like code summarization.

(RQ3) What is the benefit of using BPE tokenization for vulnerability repairs?

Approach. To answer this RQ, we aim to investigate the impact of the tokenization component for vulnerability repairs. To do so, for each approach, we alter only the tokenizer for each of the vulnerability repair approaches. Specifically, we extend our experiment to systematically evaluate the following six variants of DL-based vulnerability repair approaches, i.e., 2 tokenizers (subword tokenizer and word-level tokenizer) × 3 model architectures (VulRepair, CodeBERT, VRepair).

- ***Subword Tokenizer + CodeT5 (our VulRepair):*** BPE tokenizer with a CodeT5 model.
- ***Word-level Tokenizer + CodeT5:*** Word-level tokenizer with a CodeT5 model.
- ***Subword Tokenizer + Vanilla Transformer:*** BPE tokenizer with a Encoder-Decoder Transformer model.
- ***Word-level Tokenizer + Vanilla Transformer (VRepair):*** Word-level tokenizer with a Encoder-Decoder Transformer model and a copy mechanism for the OOV problem.
- ***Subword Tokenizer + CodeBERT (Original CodeBERT):*** BPE tokenizer with a CodeBERT model.
- ***Word-level Tokenizer + CodeBERT:*** Word-level tokenizer with a CodeBERT model.

Finally, we evaluate the accuracy of these variants using the same evaluation measure (i.e., % Perfect Predictions).

Result. Figure 4 presents the experimental results of the benefits of using BPE tokenization for vulnerability repairs.

Regardless of the model architectures, the BPE subword tokenization improves the percentage of perfect predictions by 9%-14% for vulnerability repair approaches. Figure 4 shows that the use of BPE subword tokenization improves %Perfect Predictions by 9% (44% − 35%) for VulRepair, 11% (34% − 23%) for VRepair, and 14% (31% − 17%) for CodeBERT. These results highlight the substantial benefits of using BPE tokenization for vulnerability repair approaches, addressing the Limitation ② of VRepair [13]. Nevertheless, our approach (BPE+CodeT5) is still top-performing for vulnerability repairs, which is 21% (44% − 23%) better than VRepair.

(RQ4) What are the contributions of the components of our VulRepair?

Approach. To answer this RQ, we aim to investigate the contribution of each component within VulRepair (Pre-training+BPE+T5) by examining the model accuracy of our VulRepair when each component is varied, comparing with a basic T5 (No Pre-training+Word-level+T5). Specifically, we extend our experiment to systematically evaluate the following four variants of T5-based vulnerability repair approaches, i.e., 2 pre-training strategies (pre-training, no pre-training) × 2 tokenizers (subword-level, word-level):

- ***Pre-training + BPE + T5 (VulRepair):*** A pre-trained T5 model with a BPE tokenizer.
- ***Pre-training + Word-level + T5:*** A pre-trained T5 model with a word-level tokenizer.
- ***No Pre-training + BPE + T5:*** A non-pre-trained T5 model with a BPE tokenizer.

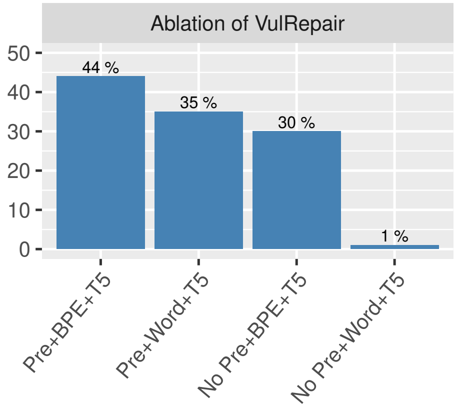

Figure 5: (RQ4) The ablation study result of VulRepair. (↗) Higher %Perfect Predictions = Better.

- ***No Pre-training + Word-level + T5:*** A non-pre-trained T5 model with a word-level tokenizer.

Similarly, we evaluate the accuracy of these variants using the same evaluation measure (i.e., % Perfect Predictions).

Result. Figure 5 presents the ablation study to evaluate the contributions of the components of our VulRepair.

The pre-training component of our VulRepair is the most important component. Within our VulRepair, the pre-training component contributes to 14% of the %Perfect Prediction. When comparing Pre+BPE+T5 and NoPre+BPE+T5 where the Pre-training component is eliminated, we observe a performance decrease from 44% to 30%, accounting for 14%. Within our VulRepair, the BPE component contributes to 9% of the %Perfect Prediction. When comparing Pre+BPE+T5 and Pre+Word+T5 where the BPE component is changed to the word level, we observe a performance decrease from 44% to 35%, accounting for 9%. Nevertheless, without a proper design of T5 architecture for our VulRepair, the performance decreased from 44% to 1%. This finding highlights that designing an NMT-based automated vulnerability repair approach is a challenging task, which requires a deep understanding of modern Transformer architectures to achieve the highest possible percentage of perfect predictions.

6 DISCUSSION

In this section, we perform additional analysis to further discuss the results of our VulRepair approach and provide some recommendations for future researchers.

6.1 What Types of CWEs that Our VulRepair Can Correctly Repair?

CWE (Common Weakness Enumeration) is a list of vulnerability weaknesses in software that can lead to security issues with its severity of risk, providing guidance to organizations and security analysts to best secure their software systems. To better understand the significance of our VulRepair on the practical usage scenarios, we perform a further investigation to better understand the Top-10 CWEs that can be correctly repaired by our VulRepair and

Table 2: (Discussion) The % Perfect Predictions of our VulRepair for the Top-10 Most Dangerous CWEs.

Rank	CWE Type	Name	%PP	Proportion
1	CWE-787	Out-of-bounds Write	30%	16/53
2	CWE-79	Cross-site Scripting	0%	0/1
3	CWE-125	Out-of-bounds Read	32%	54/170
4	CWE-20	Improper Input Validation	45%	68/152
5	CWE-78	OS Command Injection	33%	1/3
6	CWE-89	SQL Injection	20%	1/5
7	CWE-416	Use After Free	53%	29/55
8	CWE-22	Path Traversal	25%	2/8
9	CWE-352	Cross-Site Request Forgery	0%	0/2
10	CWE-434	Dangerous File Type	-	-
		TOTAL	38%	171/449

the Top-10 most dangerous CWEs.[1] The Top-10 most dangerous CWEs are the most common and impactful issues experienced over the previous two calendar years. Such weaknesses are dangerous because they are often easy to find, exploit, and can allow adversaries to completely take over a system, steal data, or prevent an application from working.

Our VulRepair can correctly repair 38% of the vulnerable functions affected by the Top-10 most dangerous CWEs (see Table 2). We find that VulRepair achieves %Perfect Predictions as much as 53% for CWE-416 (Use After Free), 45% for CWE-20 (Improper Input Validation), and 33% for CWE-78 (OS Command Injection). Figure 6 shows that our VulRepair achieves 100% perfect predictions for the following CWEs (i.e., CWE-755, CWE-706, CWE-326, CWE-667, CWE-369, CWE-77, CWE-388, CWE-436, CWE-191). However, %Perfect Predictions still vary from 0% to 100%, depending on the CWE types in the dataset. That means the CWE types that achieve perfect predictions may not necessarily be the majority of CWEs in the dataset. Thus, we further analyze the % perfect predictions according to the majority of the CWEs in the dataset. We find that there exist many CWE types that our VulRepair cannot correctly generate vulnerability repairs (e.g., CWE-639, CWE-354, CWE-522) (see Figure 6). We find that these CWE types are rare in our datasets with less than 5 functions, indicating that our VulRepair still cannot accurately repair for some types of rare vulnerabilities. Thus, *future researchers should further explore techniques to handle rare vulnerabilities (i.e., the low proportion of vulnerabilities in the training and testing dataset).*

6.2 How Do the Function Lengths and Repair Lengths Impact the Accuracy of Our VulRepair?

Although our VulRepair can correctly generate a considerable number of vulnerability repairs ($\frac{745}{1,706}$) for various types of CWEs, there is a large number of 961 vulnerable functions that cannot be correctly generated. Thus, we perform a further investigation to analyze the accuracy of our VulRepair with respect to the repair length and the function length.

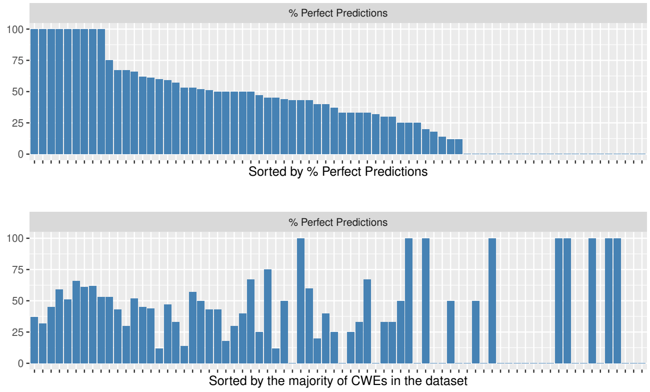

Figure 6: (Discussion) The %Perfect Predictions (y-axis) of our VulRepair according to each type of CWE (x-axis, sorted by % perfect predictions and sorted by the majority of CWEs in the dataset). Detailed statistics can be found in Appendix.

The accuracy of our VulRepair depends on the size of the vulnerable functions and its difficulty to repair. Table 3 shows that our VulRepair is most accurate for vulnerable functions that have less than 500 tokens and less than 20 repair tokens. Our VulRepair achieves the %Perfect Prediction of 64%-77% for for vulnerable functions with less than 500 tokens, but the %Perfect Prediction is substantially decreased to 32% for vulnerable functions with greater than 500 tokens. The performance decrease for large functions (500+ tokens) has to do with the window size of the T5 architecture (i.e., limited to 512 tokens). For any vulnerable functions with greater than 512 tokens, such extra tokens will be truncated and will not be processed and learned by the models, leading to a negative impact on the accuracy of our VulRepair. Thus, *future researchers should further explore techniques that can handle larger functions (i.e., the functions with more than 512 tokens).*

In addition, the repair difficulty (measured by #repair tokens in the vulnerability repair) is also impacting the accuracy of our VulRepair. We find that our VulRepair achieves the %Perfect Prediction of 63%-77% for vulnerable repairs with less than 10 repair tokens, but the %Perfect Prediction is substantially decreased to below 60% for vulnerable repairs with greater than 10 repair tokens. Thus, *future researchers should further explore techniques that can handle difficult repairs (i.e., repairs with more than 20 repair tokens).*

6.3 How Does the Complexity of the Input Functions Impact the Accuracy of Our VulRepair?

It is possible that highly-complex vulnerable functions may be more difficult to generate repairs by our VulRepair than the others. Thus, we further investigate the accuracy of our VulRepair for various degrees of the complexity of the input functions. To do so, we measure the program complexity using a Cyclomatic Complexity measure. Cyclomatic Complexity (CC) is a quantitative measure of the number of linearly independent paths through a program's source code.

[1] https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

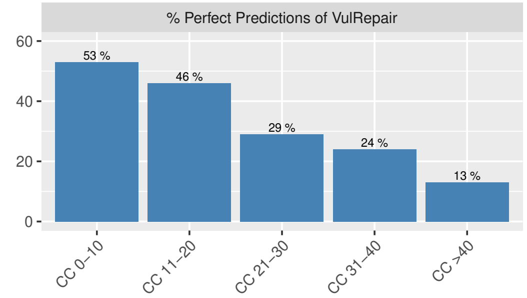

Figure 7: (Discussion) The accuracy of our VulRepair for various ranges of the Cyclomatic Complexity of the input vulnerable functions in the testing set. (↗) Higher % Perfect Predictions = Better.

Table 3: (Discussion) The % Perfect Predictions of our VulRepair according to the function length and the repair length.

		Function Lengths (#Tokens)					
		0-100	101-200	201-300	301-400	401-500	500+
Repair Lengths (#Repair Tokens)	0-10	77%	64%	75%	76%	67%	32%
	11-20	63%	56%	59%	43%	33%	32%
	21-30	50%	55%	56%	65%	56%	33%
	31-40	48%	53%	57%	42%	56%	15%
	41-50	54%	61%	53%	45%	20%	30%
	50+	48%	24%	32%	28%	16%	6%

Our VulRepair achieves a higher accuracy for less-complex input functions than highly-complex input functions. Figure 7 presents the accuracy of our VulRepair under different ranges of Cyclomatic Complexity (CC). We find that when the input functions are less complex (i.e., the CC is less than 20), our VulRepair can achieve %PP of 53% and 46%, which are better than the average accuracy (i.e., %PP of 44%). However, the accuracy of our VulRepair merely achieves %PP of 29%, 24%, and 13% when the input functions are more complex (i.e., the CC is higher than 20). Thus, *future researchers should further explore techniques that can handle highly-complex functions to further improve the accuracy of the NMT-based vulnerability repair models.*

6.4 How Well Can Our VulRepair Handle the OOV Problem of Vulnerability Repairs?

To better understand how well can BPE handle the OOV problem of vulnerability repairs, we perform a further investigation to analyze the pairs of vulnerable function-repair where new tokens never appeared in the vulnerable function but appear in the vulnerable repairs (i.e., out-of-vocabulary). Among the 1,706 pairs in the testing dataset, we find that 37% of them (627 pairs) have new tokens that appear in the vulnerable repair, but never appeared in the vulnerable function. This means that the OOV problem is accounting for 37% of the testing dataset, indicating that the size of the OOV problems is considerably large and important. Among the 627 pairs with OOV problems, we find that 37% of the vulnerable functions ($\frac{234}{627}$) can be correctly and automatically repaired by our VulRepair approach, indicating that our VulRepair can accurately generate vulnerability repairs when new are introduced in the repair version. On the other hand, VRepair which leverages the copy mechanism cannot accurately generate any vulnerability repairs when new tokens are introduced in the repair version, since the copy mechanism cannot reuse tokens that never appear in the vulnerability function to the vulnerability repairs. This finding highlights the importance of using BPE to handle the OOV problem for vulnerability repairs.

Nevertheless, we find that the correct vulnerability repairs (37%) often have new tokens ranging from 1 to 12 new tokens (avg=1.59), while the incorrect vulnerability repairs (the remaining 63%) have a relatively higher number of new tokens ranging from 1-100 new tokens (avg=4.85). This finding confirms that the correct generation of vulnerability repairs depends on its complexity and difficulty (#new tokens). Thus, *future researchers should further explore other techniques to address the OOV problem for more difficult types of vulnerability repairs.*

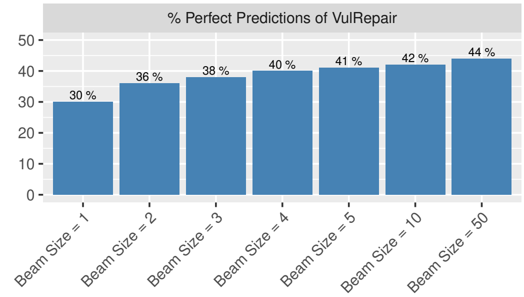

Figure 8: (Discussion) The performance of our VulRepair with different values of beam size. (↗) Higher % Perfect Predictions = Better.

6.5 How Well Can Our VulRepair Generate Perfect Repairs Given Different Values of Beam Size?

To ensure a fair comparison with the previous VRepair approach [13], we adopt a Beam Width $\beta = 50$ that returns 50 repair candidates. However, security analysts may spend a huge amount of effort to manually inspect such a large number of 50 repair candidates, which may hinder its adoption in practice. Similar to previous works [15, 52], we further investigate the accuracy of our VulRepair at the top-1 to top-5, and top-10 candidates, which is more practical and requires less manual inspection.

Figure 8 presents the accuracy of our VulRepair under different values of Beam Width (β). When analyzing the top-1 accuracy (Beam Width $\beta = 1$), we find that our VulRepair is still accurate, achieving a %PP of 30%. This means that our VulRepair can correctly repair 513 vulnerable functions out of 1,706 testing instances when only evaluating the best repair candidate recommended by VulRepair. On the other hand, when increasing the k candidates, the top-k accuracy of our VulRepair still increased, e.g., 41% to 44% when changing the k candidates from 5 to 50.

7 RELATED WORK

NMT-based APR. Researchers proposed to leverage various Neural Machine Translation (NMT) approaches for Automated Program Repair (APR). For example, Chen *et al.* [14] proposed SequenceR, which is a vanilla version of Transformer with copy mechanism to handle OOV. Jiang *et al.* [24] proposed CURE, which is a GPT architecture [41], pre-training on source code. Mashhadi and Hemmati [33] leveraged CodeBERT [19] to repair Java bugs automatically. Tufano *et al.* [53] leveraged an RNN-based NMT model to automatically generate repairs in the context of code review. Lutellier *et al.* [32] proposed CoCoNuT, which is a CNN-based NMT model to generate bug-fixes. Li *et al.* [29] proposed DLFix, which is a tree-based RNN architecture to generate bug-fixes. Thongtanunam *et al.* [52] proposed AutoTransform, which is a vanilla version of Transformer with a BPE tokenization to handle OOV. While NMT-based APR shares a similar concept of using NMT for a Code→Code task, NMT-based APR approaches [14, 24, 29] are designed to learn to generate a patch for bug-fixing purposes, which may not be related to other specific types of bugs like software vulnerabilities. In addition, such NMT-based APR approaches are designed to generate a patch that satisfies test cases.

Different from NMT-based APR, VulRepair aims to automatically generate vulnerability repairs that are exactly matched with the human-written repairs (i.e., the fix version).

NMT-based AVR. Researchers proposed NMT-based Automated Vulnerability Repair (AVR) approaches. For example, Chen *et al.* [13] proposed VRepair which leverages a word-level tokenizer and a vanilla Transformer model. Similar to VRepair, Chi *et al.* [15] proposed SeqTrans which relies on the same components as VRepair. Both VRepair and SeqTrans leverage a word-level tokenizer, however, Chen *et al.* [13] leverage the copy mechanism and Chi *et al.* [15] leverage code normalization to solve the OOV problem.

Different from NMT-based AVR, our VulRepair is the first to leverage a T5 architecture with BPE and the pre-training of a large code corpus for automated vulnerability repairs. Our systematic and comprehensive evaluations also demonstrate the substantial accuracy improvement of our approach to address various limitations of VRepair [13], highlighting the significant advancement of the NMT-based Automated Vulnerability Repair literature. Additional investigation in the Discussion section also provide recommendations for future researchers.

8 THREATS TO VALIDITY

As for any empirical study, there are various threats to the validity of our results and conclusions.

Threats to internal validity are related to the degree to which our study minimizes systematic error. Our VulRepair consists of various hyperparameter settings (i.e., number of hidden layers, number of attention heads, and learning rate). Prior studies raise concerns that different hyperparameter settings may have an impact on the evaluation results, especially, for defect prediction models [49, 51]. However, finding an optimal hyperparameter setting can be very expensive given the large search space of the Transformer architecture. Instead, the goal of our work is not to find the best hyperparameter setting, but to fairly compare the accuracy of our approach with the existing baseline approaches. Thus, the accuracy reported in the paper is served as a lower bound of our approach, which can be even further improved through hyperparameter optimization. To mitigate this threat, we report the hyperparameter settings in the replication package to aid future replication studies.

Threats to external validity are related to the degree to which our findings can be generalized to and across other vulnerabilities and projects. Our VulRepair approach is evaluated on the CVEFixes [10] and Big-Vul [18] corpus, which consists of 8,482 vulnerability repairs from 180+ different CWEs. However, the results of VulRepair do not necessarily generalize to other CWEs and other datasets. Thus, other datasets can be explored in future work.

Recently, Liu *et al.* [30] suggested that the accuracy of an automated program repair approach should not be solely evaluated based on a perfect match. Instead, other measures should also be considered, e.g., the number of semantically correct repairs and the number of plausible patches. However, these two measures require test cases for evaluating whether the repairs can successfully pass the test cases or not. Unfortunately, there exists no test cases available in the experimental dataset that we used in this paper. Thus, both measures cannot be evaluated. Nevertheless, future researchers should create new vulnerability repair datasets where such repairs are reproducible and test case information is available.

9 CONCLUSION

In this paper, we propose VulRepair, a T5-based automated software vulnerability repair approach. Through an extensive evaluation, we conclude that our VulRepair is considerably 13%-21% more accurate than VRepair and CodeBERT, highlighting the substantial advancement of NMT-based Automated Vulnerability Repairs. Importantly, our VulRepair can accurately repair as many as 745 out of 1,706 real-world well-known vulnerabilities. Importantly, we find that our VulRepair can correctly repair 38% of the vulnerable functions related to the Top-10 most dangerous CWEs, e.g., CWE-416 (Use After Free), CWE-20 (Improper Input Validation), and CWE-78 (OS Command Injection), demonstrating the practicality and significance of our VulRepair for generating vulnerability repairs, helping under-resourced security analysts on fixing vulnerabilities.

Our additional analysis discovers important findings, leading to many open research challenges that future researchers should explore (e.g., to handle larger functions, to handle rare types of vulnerabilities, to handle difficult repairs with many new tokens).

ACKNOWLEDGMENT

Chakkrit Tantithamthavorn was supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941).

DATA-AVAILABILITY STATEMENT

To support the open science community, we publish the studied dataset, scripts (i.e., data processing, model training, and model evaluation), and experimental results in GitHub (https://github.com/awsm-research/VulRepair) and Zenodo [8].

REFERENCES

[1] [n.d.]. 25+ cyber security vulnerability statistics and facts of 2021. https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/.

[2] [n.d.]. Checkmarx. https://checkmarx.com/.

[3] [n.d.]. Cppcheck. https://cppcheck.sourceforge.io/.

[4] [n.d.]. Fixing Vulnerabilities Costs 100x More If You Don't Understand the Weakness. https://medium.com/@CWE_CAPEC/fixing-vulnerabilities-costs-100x-more-if-you-dont-understand-the-weakness-c3877f68d8a6.

[5] [n.d.]. Flawfinder. https://dwheeler.com/flawfinder/.

[6] [n.d.]. National Vulnerability Database. https://nvd.nist.gov/.

[7] [n.d.]. RATS. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[8] [n.d.]. VulRepair: A T5-Based Automated Software Vulnerability Repair. https://doi.org/10.5281/zenodo.7080271.

[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[10] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[11] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2013. Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers". (2013).

[12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[13] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2021. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* (2021).

[14] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[15] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2022. Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering* (2022).

[16] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[17] M. Dowd, J. McDonald, and J. Schuh. 2006. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional.

[18] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *in Proceedings of the International Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020).

[20] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

[21] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE.

[22] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[23] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[24] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[25] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2020. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)* (2020), To Appear.

[26] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. 2021. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. To Appear.

[27] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *in Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.

[28] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: An Explainable Just-In-Time Defect Prediction Bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1336–1339.

[29] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 602–614.

[30] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817.

[31] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *in Proceedings of the International Conference on Learning Representations (ICLR)* (2017).

[32] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis (ISSTA)*. 101–114.

[33] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *in Proceedings of the International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.

[34] Patrick J Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. 2018. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering* 23, 3 (2018), 1383–1421.

[35] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.

[36] Van Nguyen, Trung Le, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. 2021. Information-theoretic Source Code Vulnerability Highlighting. In *International Joint Conference on Neural Networks (IJCNN)*.

[37] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung. 2019. Deep Domain Adaptation for Vulnerable Code Function Identification. In *The International Joint Conference on Neural Networks (IJCNN)*.

[38] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*.

[39] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* (2022).

[40] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 407–418.

[41] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research (JMLR)* (2019).

[43] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[44] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 393–410.

[45] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *in Proceedings of the Association for Computational Linguistics (ACL)* (2015).

[46] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *in Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)* (2018).

[47] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems (NeurIPS)* 27 (2014).

[48] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2021. Actionable analytics: Stop telling me what it is; please tell me what to do. *IEEE Software* 38, 4 (2021), 115–120.

[49] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th international conference on software engineering (ICSE)*. 321–332.

[50] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)* (2017).

[51] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering (TSE)* 45, 7 (2018), 683–711.

[52] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: Automated Code Transformation to Support Modern Code Review Process. In *Proceedings of the ACM/IEEE 44nd International Conference on Software Engineering (ICSE)*.

[53] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems (NeurIPS)* 30 (2017).

[55] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1943–1958.

[56] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *in Proceedings of the International Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2021).

[57] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 32–41.

[58] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE)*. IEEE, 171–180.

[59] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering (TSE)* (2020).

[60] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[61] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[62] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 327–337.

[63] Zhe Yu, Christopher Theisen, Laurie Williams, and Tim Menzies. 2019. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering* (2019).

[64] Wenyuan Zeng, Wenjie Luo, Sanja Fidler, and Raquel Urtasun. 2016. Efficient summarization with read-again and copy mechanism. *arXiv preprint arXiv:1611.03382* (2016).

[65] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems (NeurIPS)* (2019).