Revisiting VulRepair: A Reproduction and Evaluation of T5-based Software Vulnerability Repair Models

Yushika Jhundoo
University of Ottawa
Ottawa, Canada
pjhun035@uottawa.ca

Rina Osman
University of Ottawa
Ottawa, Canada
rosma012@uottawa.ca

Olena Naim
University of Ottawa
Ottawa, Canada
onaim017@uottawa.ca

Professor Paria Shirani
University of Ottawa
Ottawa, Canada
pshirani@uottawa.ca

ABSTRACT

This study reproduces and evaluates VulRepair, a T5-based neural machine translation (NMT) model for automated vulnerability repair. Originally, VulRepair was proposed as an NMT-based repair system leveraging pre-training and BPE tokenization to overcome key limitations in earlier models, such as VRepair. These limitations include a small training dataset, word-level tokenization, and a basic transformer architecture.

In our work, we re-implement all ten VulRepair model variants and conduct a comprehensive evaluation to address four key research questions (RQs): model accuracy (RQ1), the impact of pre-training (RQ2), the benefits of BPE tokenization (RQ3), and the contribution of each VulRepair component (RQ4). To ensure fairness, we evaluate all models on a deduplicated CVEFixes dataset, removing overlaps between training and test sets that can inflate performance.

Our findings confirm the importance of VulRepair's architectural choices but show that deduplication affects reported accuracy in inconsistent ways. While some models perform worse without duplicates, others improve. Notably, models M1 and M2 were not impacted by deduplication, as their data did not contain overlap. On this noise-reduced dataset, our best model (M1) still achieved a high perfect prediction rate of 46.01%, aligning with the 44% reported in the original paper. Removing both pre-training and BPE dropped performance to 0.35%, confirming their critical roles.

This reproduction study contributes insights into the reproducibility of VulRepair and shows how dataset quality influences the evaluation of vulnerability repair systems.

KEYWORDS

Vulnerability Repair, T5, Transformer, Dataset Deduplication, Neural Machine Translation, Pre-trained Language Models, Reproducibility, Software Engineering

1 INTRODUCTION

Software vulnerabilities are weaknesses in software systems that attackers can exploit to cause harm or steal sensitive information. Vulnerabilities are one of the leading causes of cybercrime, which is projected to cost the global economy over $10.5 trillion annually by 2025 [20]. In the past decade, the number of reported vulnerabilities has increased fivefold [16]. While many machine learning techniques have been developed to detect vulnerabilities, they often stop at identification, leaving the manual task of fixing vulnerabilities to human developers.

To help address this challenge, researchers have begun exploring NMT to automate vulnerability repair. A recent approach, VulRepair, introduced a T5-based NMT model to automatically generate fixes for buggy code [7]. The authors attribute its strong performance to three main design decisions: using a pre-trained CodeT5 model, applying Byte-Pair Encoding (BPE) tokenization, and training on the CVEFixes dataset, a large collection of vulnerability patches.

VulRepair was developed to overcome several limitations of earlier systems like VRepair [7]. First, VRepair was trained on a relatively small corpus of 23,607 C/C++ functions, which limited its ability to learn meaningful representations. Second, it relied on word-level tokenization and a copy mechanism to handle Out-Of-Vocabulary (OOV) tokens. This constrained the model's ability to introduce new repair tokens not seen in the original buggy code. Finally, VRepair used a standard encoder-decoder Transformer with absolute positional encodings, reducing its capacity to understand relative code structure during the repair process.

In this work, we reproduce the VulRepair study to evaluate the reliability of its results and explore how the model performs under different conditions. We use the same hyperparameters provided in the public GitHub repository, except for a smaller batch size due to limited hardware. Our source code and modified pipeline are available at[1].

Unlike the original study, we train and evaluate our models on a deduplicated version of the CVEFixes dataset. Duplicate examples in both training and test sets can lead to inflated performance metrics. By removing these duplicates, we aim to provide a more accurate and fair evaluation of model effectiveness.

We also aim to understand the contributions of each of VulRepair's core components. We replicate and analyze the following research questions:

(RQ1) What is the accuracy of VulRepair for generating software vulnerability repairs? Results. We observe lower performance compared to the original paper. On deduplicated data,

[1]VulRepair Replication Repository: https://github.com/rinaxosman/VulRepair. Developed by Yushika Jhundoo, Rina Osman, and Olena Naim at the University of Ottawa as a reproduction of the VulRepair paper.

our best model achieved a Perfect Prediction of 46.01%, compared to 44% reported in the original work. This highlights the impact of duplicates on evaluation results.

(RQ2) What is the benefit of using a pre-training component for vulnerability repair? Results. Removing pre-training caused performance to drop significantly from 46.01% to 0.64%. This confirms that pre-training is critical for the model's success.

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

Results. Switching from BPE to word-level tokenization caused a modest decrease in perfect repair rates, from 46.01% to 44.08% before deduplication. For word-level tokenization, after deduplication, the rate fell sharply to 9.28%. The advantage of BPE tokenization is generalizing the limited data available.

(RQ4) What are the contributions of each component of the VulRepair architecture? Results. Our ablation study shows that removing both BPE and pre-training drops accuracy from 46.01% to 0.35%, reinforcing the importance of both components. The pre-training component had the largest impact.

These findings show meaningful differences from the original VulRepair paper, especially when using deduplicated data. This study contributes to the discussion on reproducibility and highlights how design choices and dataset quality influence model performance. Our results also show that VulRepair has the potential to assist security analysts by suggesting high-quality repair candidates. We found that the model could generate valid patches for real-world vulnerabilities such as Use After Free and NULL Pointer Dereference. While not perfect, it significantly reduces the manual effort needed to fix vulnerable code.

The contributions of our study are as follows:

- A full reproduction of the VulRepair framework using the original implementation and configuration.
- A new evaluation using deduplicated data to test performance more fairly and reduce data leakage.
- An analysis of the impact of pre-training and tokenization strategies on model accuracy.
- An ablation study showing how each component (tokenizer, pre-training, architecture) affects results.
- Public release of our cleaned dataset, evaluation scripts, and training pipeline for future research [8].

Paper Organization. Section 2 describes the problem and limitations of prior work. Section 3 presents the VulRepair architecture. Section 4 explains the experimental setup. Section 5 reports our results. Section 6 provides further discussion. Section 7 reviews related work. Section 8 discusses threats to validity, and Section 9 concludes the paper.

2 BACKGROUND & PROBLEM MOTIVATION

Figure 1 sketches the vulnerability lifecycle: Unclosed zero-day discovery, CVE assignment and publication, and the ensuing N-day exposure window before patch adoption. With thousands of vulnerabilities reported each year, automating both their detection and repair has become a critical research frontier [20] [10].

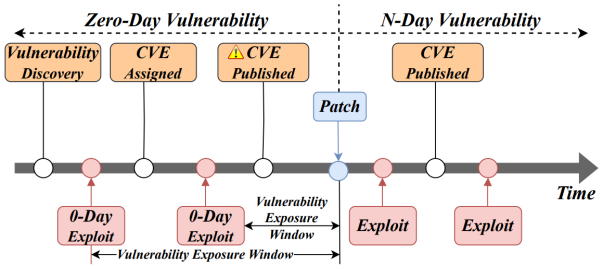

Figure 1: Vulnerability discovery timeline.[2]

The first major step in this direction was VRepair, which models vulnerability fixing as a transfer-learning problem: it learns to correct general bugs, and the model could apply that knowledge to security flaws. The three-stage pipeline of VRepair tokenizes C functions with a word-level Clang tokenizer, embeds those tokens alongside absolute positional encoding, feeds them into a six-layer encoder-decoder transformer, and finally uses beam search to produce 50 candidate fixes. Despite this approach, VRepair faced three key hurdles: a small training dataset, poor handling of out-of-vocabulary tokens, and limited modeling of token positions due to its reliance on absolute embeddings.

To address these limitations, researchers from Monash University in Sydney proposed VulRepair. VulRepair utilized large-scale code pre-training, incorporated Byte Pair Encoding (BPE) for improved token handling, the T5 architecture, and applied relative positional encoding to more accurately capture the structure of source code. These enhancements significantly improved model performance, achieving a perfect prediction rate of 44%, surpassing both VRepair and CodeBERT.

In our project, we recreate and validate VulRepair's results, motivated by the importance of reproducibility in security-critical machine learning and by our own learning goal, as students, of deepening our hands-on understanding of large language models.

3 VULREPAIR: A T5-BASED VULNERABILITY REPAIR APPROACH

This section describes the architecture of VulRepair, a T5-based NMT model designed to automatically fix software vulnerabilities by translating buggy code into repaired code. The process includes three stages: BPE-based code tokenization, an encoder-decoder architecture for repair generation, and beam search decoding.

Overview. Given a vulnerable function, the repair process begins in Step 1, where the source code is tokenized using Byte-Pair Encoding (BPE) based on a pre-trained CodeT5 tokenizer [21]. This produces subword-tokenized input for both the vulnerable code and its corresponding repair.

[2]Adapted from Li et al., *SoK: Towards Effective Automated Vulnerability Repair* [10].
[3]Adapted from the original VulRepair paper by Fu et al. [7].

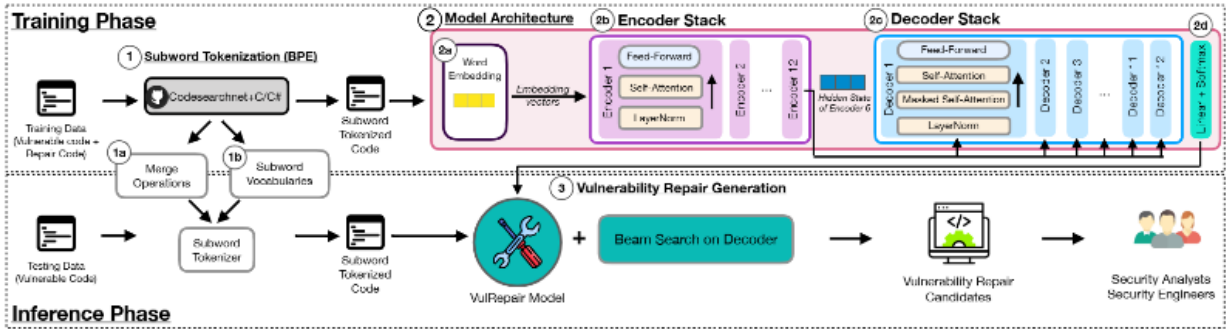

Figure 2: Architecture of the VulRepair model.[3]

3.1 Code Representation

Before being fed into the model, vulnerable functions are pre-processed using Byte-Pair Encoding (BPE) tokenization. This technique breaks down uncommon tokens into subwords while preserving commonly seen identifiers. For example, the identifier `IsValidSize` might be split into `["IsValid", "Size"]`.

The tokenizer used in VulRepair was pre-trained on the CodeSearchNet dataset and a large C/C# corpus [21], covering eight programming languages. It is further customized with special tokens like <s>, </s>, <pad>, and vulnerability markers <StartLoc>, <EndLoc>, <ModStart>, and <ModEnd> to help the model localize and learn the buggy region more effectively.

Each subword token is embedded into a 768-dimensional vector using pre-trained embeddings. Unlike earlier models like VRepair, which use absolute positional encodings, VulRepair incorporates relative positional encoding into the self-attention mechanism. This allows the model to learn how tokens relate to one another based on their position and distance in the sequence.

3.2 VulRepair Model Architecture

VulRepair uses the T5 encoder-decoder architecture [17]. The encoder has 12 stacked layers, each containing a multi-head self-attention layer and a feed-forward network, with residual connections and layer normalization applied throughout. The attention mechanism relies on query (Q), key (K), and value (V) matrices. Relative positional information is added via a matrix P, helping the machine understand local dependencies and long-range relationships. .This matrix P is added to both K and V during attention computation:

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

The decoder mirrors the encoder in architecture, with 12 layers that include masked self-attention (to prevent future token access during generation), encoder-decoder attention, and feed-forward layers. Final outputs are passed through a linear projection and softmax to generate token probabilities.

3.3 Vulnerability Repair Generation

Once the decoder outputs token probabilities, VulRepair uses beam search to generate the final fixed function. Beam search (See Appendix) keeps multiple high-probability sequences at each step, based on a defined beam width (set to 50 in the original study). Decoding continues until the end-of-sequence token </s> is produced. Each predicted sequence is compared to the ground truth fix. A prediction is considered correct if it exactly matches the target repaired code. This evaluation method is used to compute metrics like the percentage of perfect predictions.

4 EXPERIMENTAL DESIGN

4.1 Research Questions

The four research questions in this study are adopted from the original VulRepair paper [7]. We aim to reproduce and investigate each question through our experiments, with an additional focus on how deduplicating the dataset affects model performance. This allows us to validate the original findings while exploring the impact of dataset quality on accuracy and robustness.

(RQ1) What is the accuracy of VulRepair for generating software vulnerability repairs?
The original paper compared VulRepair with two baseline models: VRepair and CodeBERT. VRepair is an early NMT-based repair model, while CodeBERT is a transformer-based model pre-trained on code. This question evaluates whether VulRepair offers higher prediction accuracy than these alternatives. We reproduce this evaluation and also analyze how accuracy changes on a deduplicated version of the dataset.

(RQ2) What is the benefit of using a pre-training component for vulnerability repair?
VulRepair leverages a CodeT5 model pre-trained on a large corpus of programming and natural language data [21]. In contrast, VRepair is trained from scratch on a smaller dataset. This research question examines how much pre-training contributes to repair accuracy. In our reproduction, we further test this by disabling pre-training and comparing results on both the original and deduplicated datasets.

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

BPE tokenization helps break down rare or unseen tokens into subwords, which may help generate more accurate repairs. The original paper showed BPE outperforms word-level tokenization. We replicate this comparison and study how the effect of tokenization varies across datasets.

(RQ4) What are the contributions of each component of the VulRepair architecture?
VulRepair integrates three core components: BPE tokenization, pre-training, and a T5 encoder-decoder model. The original work conducted an ablation study to assess the contribution of each. We reproduce this ablation analysis and extend it to the deduplicated dataset to identify which components are most sensitive to data quality.

4.2 Studied Dataset

In our experiments, we use the CVEFixes dataset introduced by Bhandari et al. [3], which contains 8,482 vulnerability fixes, each represented as a pair of vulnerable and repaired C functions. The data is collected from 1,754 real-world open-source projects spanning 180 different CWE IDs, from the years 1999 to 2021.

To ensure consistency with prior work, we followed the same pre-processing pipeline used in the original VulRepair study [7]. Each input sequence includes markers that identify the vulnerable code region. The tags <StartLoc> and <EndLoc> are used to mark the beginning and end of the buggy code within the function, while <ModStart> and <ModEnd> are used in the output to mark the start and end of the repaired segment. These special tokens are added to help the tokenizer preserve the structure of the sequence and guide the model's attention toward the critical code regions during training and generation.

4.3 Experimental Setup

Dataset. Following the original VulRepair paper [7], we split the CVEFixes dataset into 70% training, 10% validation, and 20% testing.

To prevent data leakage, we ran our set-wise deduplication script (see Appendix) on the existing training, validation, and test splits of the CVEFixes dataset, removing any exact duplicate examples. Because the original random split seed was not available, we retained the published partitions and focused solely on deduplication to keep comparable results. After this step, the dataset size dropped by 28% across all three splits. Finally, we evaluated every model on the original CVEFixes and our deduplicated version, a fairer comparison, and revealed any metric inflation caused by overlapping examples.

Implementation. We built our version of VulRepair using the HuggingFace Transformers and PyTorch libraries. The pre-trained CodeT5 model and tokenizer were obtained through the Transformers API. The tokenizer had been trained on CodeSearchNet and a C/C# corpus [21].

Training Setup. All training and evaluation were conducted on the Beluga cluster (Digital Research Alliance of Canada), a supercomputer with 688 NVIDIA GPUs and 38,000 CPU cores. Each of the 10 VulRepair variants (original and deduplicated datasets) required roughly 6–9hours to train, and less than 2 hours for full inference.

For comparison, on a local workstation (no GPU), running just 1/75 of an epoch took 5 hours. We trained all 10 VulRepair variants across both original and deduplicated datasets. We reused the original model configuration and hyperparameters, including the learning rate, architecture size, and number of epochs. Only minor changes were made to the codebase to support updated dependencies or to reference our custom dataset paths.

Custom shell scripts were written for each model variant to automate training and evaluation jobs on SLURM. The model checkpoints were selected based on validation loss, not test set performance, to preserve fair evaluation. Our experiments used a beam width of 50 during inference to generate candidate sequences, consistent with the original paper.

Due to hardware constraints, we used a smaller batch size (4 instead of 8) during training to fit the models into GPU memory on Beluga.

Hyperparameters. We used the default CodeT5-base configuration: 12 encoder and 12 decoder layers, 768-dimensional hidden states, and 12 attention heads. The learning rate was set to 2e-5 with a linear scheduler. The AdamW optimizer was used for fine-tuning with cross-entropy loss between the predicted and ground-truth token distributions.

5 EXPERIMENTAL RESULTS

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

Approach. We reproduce the original VulRepair experiments to evaluate how accurately the model generates correct vulnerability repairs. As in the original study [7], we use the metric *Perfect Prediction*, which measures the percentage of repaired functions where the model output exactly matches the human-written patch. We use beam search with a width of 50 to generate multiple candidate sequences, and consider a prediction correct if any of the candidates matches the reference.

We also reference the baseline models reported in the original paper:

- **VRepair**, a Transformer encoder-decoder model trained from scratch on a smaller bug-fix dataset [4].
- **CodeBERT**, a pre-trained encoder-only Transformer fine-tuned for code-related tasks, including vulnerability repair [6].

Results. On the original (non-deduplicated) dataset, our implementation of VulRepair (Model M1) achieved a Perfect Prediction of 46.01%, outperforming the baseline results reported in the original paper (CodeBERT: 35%, VRepair: 23%). These results confirm that VulRepair's use of pre-training, BPE tokenization, and a T5 encoder-decoder architecture leads to stronger performance in repair accuracy.

Additional Insight. Models M1 and M2 were not affected by deduplication, as they contained no overlapping samples between the training and test sets. Other models in our study did show variance between the original and deduplicated datasets, indicating that some models may or may not benefit from data leakage. All in all, our results reinforce the original claim that VulRepair outperforms baseline systems.

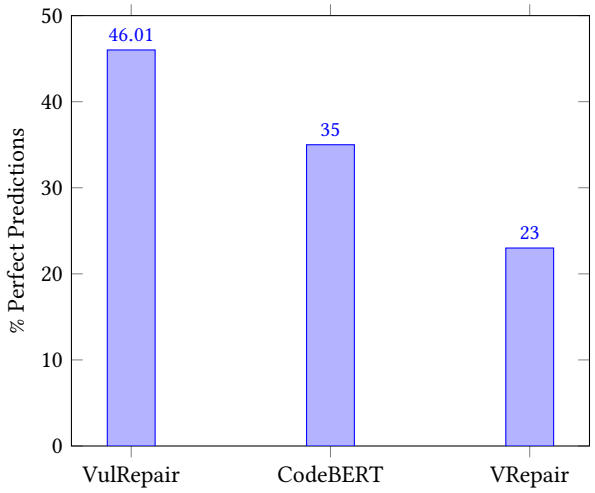

Figure 3: (RQ1) Accuracy comparison of VulRepair against baseline models using Perfect Prediction. (↑) Higher = Better.

Discussion. Figure 3 visually demonstrates the superiority of VulRepair over the two baselines. VulRepair outperforms CodeBERT by 11.01 percentage points and VRepair by 23.01 points. These margins show that a well-pretrained encoder-decoder model with proper tokenization can generate significantly more accurate repairs than both simpler Transformer models and encoder-only models.

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

Approach. This research question investigates the impact of pre-training on model performance. The original VulRepair paper showed that models pre-trained on large corpora of programming and natural language data (PL/NL) performed significantly better than models trained from scratch [7, 21]. To test this, we trained several models using the same architecture (T5 or BERT), with and without pre-trained weights.

Results. Pre-training had a large impact on model accuracy. Our best T5-based model (M1), which used pre-trained CodeT5 weights, achieved 46.01% Perfect Prediction. When we removed pre-training (Model M10), performance dropped to just 0.35%, a decrease of 45.65 percentage points. Similarly, the BERT-based model dropped from 34.95% (M2) to 1.43% (M5) when pre-training was removed, a 33.52 percentage point loss.

Conclusion. These results confirm the original paper's findings: pre-training is essential for achieving high accuracy in vulnerability repair. The pre-trained models learned useful representations from large code corpora, which allowed them to generate more accurate and context-aware repairs. The sharp performance drop in models trained from scratch highlights the value of using large-scale PL/NL data during pre-training for this task.

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

Approach.

We compared subword (BPE) and word-level tokenization across three model architectures, T5, Vanilla Transformer, and BERT, while holding all other training settings constant. To evaluate robustness, each model was assessed before and after deduplication of the CVEFixes vulnerability repair dataset (where applicable).

Results. Across all three architectures, Byte Pair Encoding (BPE) tokenization consistently outperformed word-level tokenization, both before and after deduplication of the concerned dataset:

- **T5 with BPE (M1):** 46.01% perfect repair accuracy (no duplicates were present, so this value holds after duplicates).
- **BERT with BPE (M2):** 35 % accuracy(no duplicates were present, so this value holds after duplicates).
- **T5 with word-level (M7):** dropped from 44.1 % to 9.3 % post deduplication.
  Vanilla Transformer with BPE (M8): improved from 33 % to 35 % after removing duplicate examples.
- **BERT with word-level (M9):** fell from 12 % to 3.2 % after deduplication.

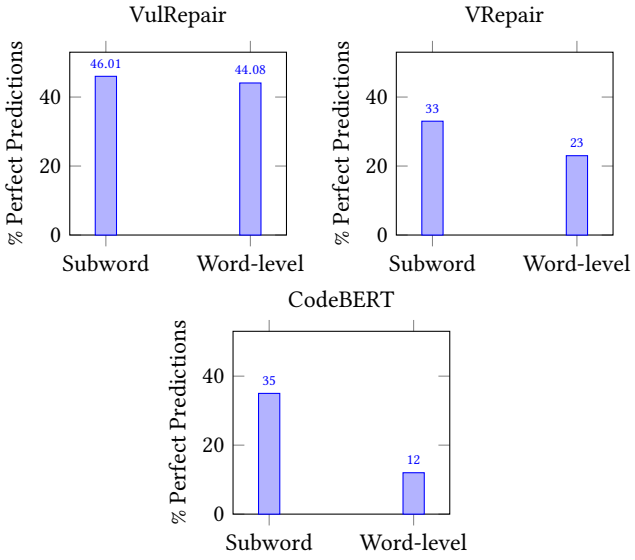

Figure 4: (RQ3) Subword (BPE) vs Word-level tokenizer performance for each model. (↑) Higher %Perfect Prediction = Better.

Further Discussion. We also evaluated the three models in CVEFixes fine-tuned deduplication dataset. BPE-based variants are unchanged (e.g., VulRepair remains at 46.0% perfect repairs), while the vanilla Transformer even ticks up slightly (33% 35%), and word-level BERT drops from 12% 3. 2% (see Table 2). These results reinforce the initial hypothesis that BPE is robust to duplicate leakage.

Conclusion (RQ3).

Our results demonstrate a clear advantage for BPE tokenization across all architectures. These steep drops for word-level tokenizers

underscore their inability to decompose words into meaningful tokens for the machine, causing them to overfit repeated tokens. BPE's splitting handles mixed-case and underscore conventions, mitigates OOV issues, and prevents metric inflation due to duplicate leakage, directly answering RQ.3.

5.4 (RQ4) What are the contributions of the components of 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 **VulRepair** system when each component is varied, comparing with a baseline T5 model (No Pre-training + Word-level + T5). Specifically, we 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.
- **No Pre-training + Word-level + T5**: A non-pre-trained T5 model with a word-level tokenizer.

All variants were evaluated using the same metric: **% Perfect Predictions**.

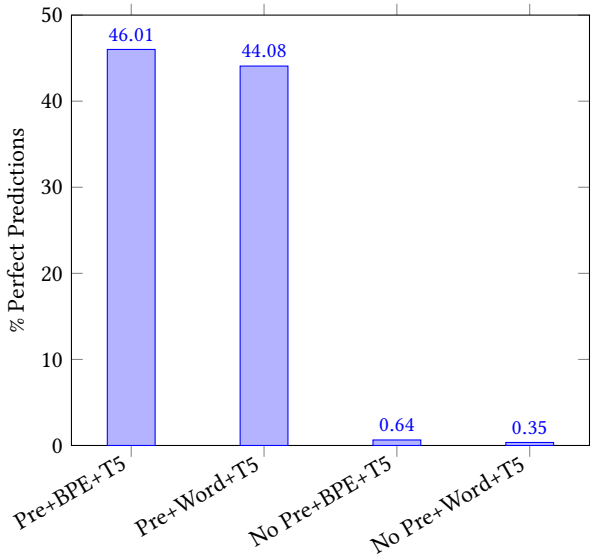

Figure 5: (RQ4) The ablation study result of VulRepair. (↑) Higher %Perfect Predictions = Better.

Result. Figure 5 presents the ablation study conducted on the deduplicated dataset. The results show that:

- **Pre-training is the most impactful component.** Comparing Pre+BPE+T5 to No Pre+BPE+T5 leads to a performance drop from 46.01% to 0.64% (↓ 45.37 points).
- **Tokenization also contributes.** Switching from BPE to Word-level (Pre+BPE+T5 vs Pre+Word+T5) drops performance slightly from 46.01% to 44.08% (↓ 1.93 points).
- **Removing both components causes a severe drop.** No Pre+Word+T5 achieves only 0.35% prediction accuracy.

These findings highlight that both pre-training and BPE tokenization are critical to VulRepair's success. Figure 5 presents the ablation study conducted on the deduplicated dataset to evaluate the contributions of each component in VulRepair. The results indicate that the pre-training component is the most impactful. When comparing the models with and without pre-training while keeping the BPE tokenizer fixed (Pre+BPE+T5 vs. No Pre+BPE+T5), the% Perfect Predictions dropped from 46.01% to 0.64%, revealing a performance loss of approximately 45.37 percentage points.

The tokenization strategy also contributes meaningfully to the model's effectiveness. Changing the tokenizer from BPE to word-level while keeping pre-training enabled (Pre+BPE+T5 vs. Pre+Word+T5) led to a slight reduction from 46.01% to 44.08%, a drop of 1.93 percentage points.

Most notably, in the absence of both pre-training and BPE tokenization (No Pre+Word+T5), the performance plummets to just 0.35%, emphasizing the necessity of both components. These findings highlight that designing a robust Transformer-based automated vulnerability repair system like VulRepair demands not only architectural depth, but also careful attention to pre-training and tokenization strategies in order to achieve high prediction accuracy.

6 DISCUSSION

In this section, we discuss key findings and observations based on the original VulRepair study [7]. Although we did not run additional experiments to investigate these points, we summarize what we learned from the reported results and offer insights for future work.

6.1 What types of CWEs can VulRepair repair accurately?

Common Weakness Enumeration (CWE) is a classification system developed to categorize software vulnerabilities based on common patterns and causes [15]. Each vulnerability in the CVEFixes dataset is labeled with a CWE identifier. According to the original paper [7], VulRepair achieved strong results on several high-impact CWEs, including Use After Free (CWE-416), Improper Input Validation (CWE-20), and OS Command Injection (CWE-78). For example, VulRepair correctly repaired 53% of CWE-416 samples, and 45% of CWE-20 samples.

However, the model's accuracy varied widely depending on the CWE. Some CWEs had 0% Perfect Prediction, such as CWE-79 (Cross-Site Scripting) and CWE-352 (Cross-Site Request Forgery). The paper attributes this to class imbalance: rare CWEs were underrepresented in the training data, making it difficult for the model to learn their patterns. Future work may benefit from data augmentation or re-sampling strategies to improve performance on rare or underrepresented vulnerability types.

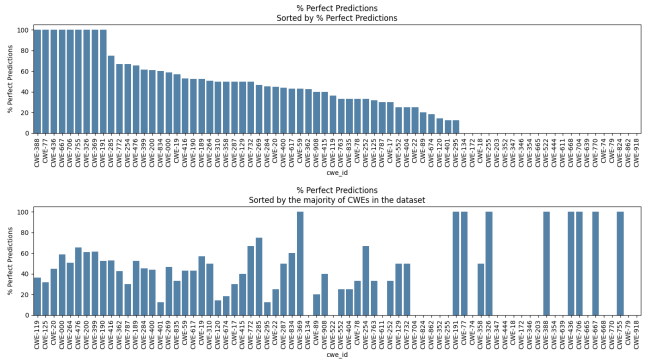

Figure 6: (Discussion) The %Perfect Predictions (y-axis) of our VulRepair according to each type of CWE.

6.2 How do function and repair lengths affect performance?

The authors of VulRepair observed that function and repair lengths significantly impacted the model's performance. Functions with fewer than 500 tokens had the highest accuracy, with up to 77% Perfect Prediction when the repair required fewer than 10 tokens [7]. However, for functions longer than 500 tokens or repairs with more than 20 tokens, accuracy dropped to around 32% or lower.

This limitation stems from the model's fixed input length. CodeT5, like most transformer-based models, has a maximum input size of 512 tokens [17]. Any code beyond that is truncated, leading to loss of important context. For more complex or longer functions, this could cut off the vulnerable code or its surrounding logic, negatively impacting repair quality. Addressing this may involve techniques like hierarchical encoding, long-context transformers [2], or segmentation strategies.

6.3 How does function complexity affect VulRepair's accuracy?

Function complexity is another important factor in vulnerability repair. The VulRepair paper measured complexity using Cyclomatic Complexity (CC), which reflects the number of independent execution paths in a function [14]. The authors found that VulRepair performed best on functions with low CC. Specifically, it achieved 53% Perfect Prediction for functions with CC between 0 and 10, compared to just 13% for functions with CC above 40 [7].

This suggests that simpler control flow and fewer branching statements make it easier for the model to understand the vulnerable context and generate a suitable fix. High CC values, on the other hand, may indicate deeply nested logic or complex data flows, which are harder for NMT models to handle accurately. Future work could explore combining program analysis techniques with NMT to handle structurally complex code better.

Discussion. Table 1 summarizes the token and complexity distributions of the CVEFixes dataset. The median number of tokens per vulnerable function is 280, but the upper quartile reaches 593 tokens, indicating that many functions are lengthy. Similarly, while the average repair length is 55 tokens, the median is only 24, suggesting a skewed distribution where a few patches are much longer than most.

[4]Table content from the original VulRepair paper [7].

Table 1: Descriptive statistics of the CVEFixes dataset.[4]

	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

7 RELATED WORK

In this section, we explain how the original VulRepair paper fits into previous work on automated program repair and automated vulnerability repair using neural machine translation (NMT) techniques.

7.1 NMT-based Automated Program Repair

Many researchers have used NMT models to automatically fix bugss in code. For example, SequenceR [5] uses a Transformer model with a copy mechanism to handle unknown words. CURE [9] uses a GPT-style model pre-trained on source code. Other work includes CoCoNuT [13], which uses CNNs, and DLFix [11], which uses tree-based RNNs. AutoTransform [19] also uses Transformers with BPE tokenization to handle out-of-vocabulary (OOV) issues.

These models focus on fixing general bugs and usually rely on test cases. In contrast, VulRepair is designed to generate vulnerability repairs that exactly match the correct fixed version written by a human, rather than just passing a test case.

7.2 Dynamic LLM-Agent Vulnerability Repair

Recent advances have introduced dynamic repair paradigms, such as VulDebugger by Liu et al. [12], which combines static code analysis with runtime execution tracing. Where static models like VulRepair operate solely on code patterns, VulDebugger employs LLM-guided debugging: it executes vulnerable programs, inspects runtime states (e.g., variable values, stack frames), and infers crash-free constraints to iteratively refine patches, closely mirroring developer workflows.

Evaluated on 50 real-world C vulnerabilities across diverse open-source projects, VulDebugger achieved a 60.00% success rate, dramatically outperforming static baselines, and VulRepair achieved 4.00%. For instance, in CVE-2016-3623, a divide-by-zero bug, VulRepair could not identify the origin of a zero-valued variable (vertSubSampling) passed through several functions, compared to VulDebugger traced and repaired the issue by analyzing runtime states. Similarly, in CVE-2016-10094, involving a heap overflow in libtiff, VulRepair missed the crash context entirely, while VulDebugger produced a valid root-cause diagnosis through iterative debugging and constraint comparison.

These results underscore limitations of static NMT models in handling vulnerabilities requiring runtime context (e.g., multi-frame

dataflow, memory corruption). Future iterations of VulRepair could adopt hybrid validation, such as agent-based crash reproduction or dynamic constraint checks, to improve reliability on complex CWEs while retaining its static pipeline's scalability.

8 THREATS TO VALIDITY

VulRepair shows promise for automated vulnerability repair, but it has limitations affecting its performance during inference.

One major issue is that its performance depends on how its hyperparameters are set. For example, when we lowered the batch size from 8 to 4, we saw a small increase in accuracy on our main test set, from 44% to 46% for VulRepair. However, this same change caused a huge drop in accuracy in M3 (`T5No Pretraining BPE`), going from 30% to 0.64%. This result suggests that the default hyperparameters used in the original paper may not be optimal for all model variants, and that certain configurations are sensitive to small changes. Future work could explore hyperparameter tuning to improve model robustness and performance across different setups.

Another important limitation is related to dataset quality. The original CVEFixes dataset [3] contained exact duplicates between the training and test sets, which may have inflated the performance of certain models. This represents a threat to external validity, as real-world systems are unlikely to encounter exact copies of code seen during training; for example, offices, code snippets are incomplete and messy, the technology differs in versions and dependencies, creating vulnerable situations for newer versions.[1] In contrast to the controlled, secure environment that we used.

Although VulRepair operates opaquely, offering little insight into its internal reasoning, experienced security analysts can still apply their expertise to review each suggested fix; however, risks of introducing subtle errors are present. Incorporating interpretable attention analyses or integrating automated program verification techniques could help users trust and safely deploy model-generated repairs [18].

By acknowledging these limitations: hyperparameter sensitivity, single-run evaluation, dataset artifacts, and model opacity. We clarify the scope of our findings and chart directions for future research to improve robustness, reproducibility, and trustworthiness.

9 CONCLUSION

We successfully reproduced VulRepair, evaluated all 10 model variants, and introduced deduplication to better understand the reliability of reported performance.

Our findings confirm the original study's claims. VulRepair, which combines a T5 encoder-decoder model, BPE tokenization, and pre-training on PL/NL data, outperforms baseline models like CodeBERT and VRepair. The highest performance was achieved by Model M1, with a perfect prediction rate of 46.01%. In contrast, models trained without pre-training (e.g., M10) performed significantly worse. Deduplication revealed performance drops in several models, highlighting potential overfitting in the presence of data leakage.

We observed that the choice of components (tokenizer, pre-training, and architecture) greatly impacts repair accuracy. Our experiments show that both BPE and pre-training are essential for stable and accurate performance.

Table 2 provides a high-level summary of all the 10 models across original and deduplicated datasets.

Table 2: Summary of Model Variants and Their Accuracy.

Model	Tokenizer	Pre-train	Arch.	Accuracy (%)
M1	BPE	PL/NL	T5	46.01 (no dupes)
M2	BPE	PL/NL	BERT	34.95 (no dupes)
M3	BPE	None	T5	0.64 → 1.61
M4	BPE	NL	T5	5.16 → 3.35
M5	BPE	None	BERT	11.96 → 1.43
M6	BPE	NL	BERT	1.80 → 3.00
M7	Word-level	PL/NL	T5	44.08 → 9.29
M8	BPE	None	Vanilla Transf.	33.00 → 35.00
M9	Word-level	PL/NL	BERT	12.00 → 3.20
M10	Word-level	None	T5	0.35 → 0.64

Arrows (→) indicate accuracy change due to deduplication.

Key Takeaways:

- VulRepair (M1) is the most accurate and robust model across both datasets.
- Removing pre-training causes a large drop in accuracy, especially for T5 (M3, M10).
- Deduplication reveals that some models were likely overfitting due to duplicate samples.
- Models using word-level tokenization tend to suffer larger performance degradation after deduplication.

Future Work: Future work should explore stronger generalization methods, improve robustness to rare vulnerabilities, and potentially long sequence architecture to better handle lengthy functions. [2]

10 APPENDIX

Deduplication Function

Below is the Python helper function we used to remove overlapping rows between our train/test/validation splits:

```
def remove_inter_duplicates(source_df, *target_dfs
    ):
    """
    Removes rows from source_df that are present
        in any of the target_dfs.

    Parameters:
        source_df (pd.DataFrame): The dataframe to
            clean
        *target_dfs (pd.DataFrame): Dataframes
            containing rows to remove from
            source_df

    Returns:
```

```
        pd.DataFrame: source_df with overlapping
            rows removed
    """
    if not target_dfs:
        return source_df

    # Combine target DataFrames and drop
        duplicates within them
    combined_target = pd.concat(target_dfs).
        drop_duplicates()

    # Merge to find overlapping rows
    merged = source_df.merge(combined_target,
                             on=list(source_df.
                                 columns),
                             how='left',
                             indicator=True)

    # Keep only rows unique to source_df
    cleaned_df = merged[merged['_merge'] == '
        left_only']\
                     .drop(columns='_merge')
    return cleaned_df
```

Table 3: Dataset sizes before and after deduplication

Metric	Whole Dataset	Test Set	Validation Set	Train Set
Before	8 482	1 706	839	5 937
After	6 104	1 614	713	3 777

Beam search algorithmic summary.

(1) Initialize $\mathcal{B}_0 = \{(\langle \text{sos} \rangle, 0)\}$.
(2) For $t = 0, \ldots, T_{\max} - 1$:
    - Expand each beam entry by all $v \in \mathcal{V}$.
    - Compute scores s_{t+1}.
    - Prune to top B to form $\mathcal{B}_{t+1}$.
    - If all hypotheses end in </s>, break.
(3) Return the highest-scoring hypothesis in $\mathcal{B}_{\leq T}$.

Beam width choice. We set $B = 50$ to match the original study, balancing search breadth against computational cost.

REFERENCES

[1] [n. d.]. *Outdated code snippets from Stack Overflow jeopardise software security.* https://cispa.de/en/jallow-stackoverflow

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. In *arXiv preprint arXiv:2004.05150.*

[3] Udit Bhandari, Lechen Zhu, Baishakhi Ray, and Michael Pradel. 2021. CVEfixes: Automated Collection of Vulnerability-Fixing Commits for Open-Source Software. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR).* IEEE, 402–406. https://doi.org/10.1109/MSR52588.2021.00052

[4] Zimin Chen, Yao Liu, Shuhao Liu, Anh Tuan Nguyen, and Lin Tan. 2021. VRepair: Fixing Vulnerabilities with a Code-aware Neural Model. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* ACM, 307–319. https://doi.org/10.1145/3460319.3464827

[5] Zimin Chen and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 631–642. https://doi.org/10.1145/3338906.3338911

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.emnlp-main.132

[7] 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).* ACM, New York, NY, USA, 13. https://doi.org/10.1145/3540250.3549098

[8] Yushika Jhundoo, Rina Osman, and Olena Naim. 2024. VulRepair Replication: Reproducing and Evaluating T5-based Vulnerability Repair. https://github.com/rinaxosman/VulRepair. University of Ottawa, Reproduction Study of VulRepair (ASE 2023).

[9] Shanqing Jiang, Ziyuan Ren, Xinran Wang, and Yinxing Sun. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE).* IEEE/ACM, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[10] Ying Li, Faysal Hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. 2025. SoK: Towards Effective Automated Vulnerability Repair. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security).* Seattle, WA. https://gangw.cs.illinois.edu/sec25-sok.pdf

[11] Yue Li, Junjie Wu, Shing-Chi Yan, Hailong Yin, Ziying Yang, Weiqing Li, Chao Zhang, and Baowen Xu. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ACM, 602–614. https://doi.org/10.1145/3368089.3409738

[12] Zhengyao Liu, Yunlong Ma, Jingxuan Xu, Junchen Ai, Xiang Gao, Hailong Sun, and Abhik Roychoudhury. 2025. Agent That Debugs: Dynamic State-Guided Vulnerability Repair. *arXiv preprint arXiv:2504.07634* (2025). https://arxiv.org/abs/2504.07634

[13] Thibaud Lutellier, Ying Gao, Yuwei Sui, Felix Yu, Lin Zhang, and Lei Zhao. 2020. CoCoNuT: Combining Contextual Information in Neural Machine Translation for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 101–114. https://doi.org/10.1145/3395363.3397360

[14] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320.

[15] MITRE. 2023. CWE - Common Weakness Enumeration. https://cwe.mitre.org/. Accessed: 2024-04-22.

[16] National Institute of Standards and Technology (NIST). 2023. NVD Statistics - Vulnerabilities by Year. https://nvd.nist.gov/vuln/search/statistics. NVD Vulnerability Stats.

[17] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. In *Journal of Machine Learning Research*, Vol. 21. 1–67.

[18] Xinyu She, Yue Liu, Yanjie Zhao, Yiling He, Li Li, Chakkrit Tantithamthavorn, Zhan Qin, and Haoyu Wang. 2023. Pitfalls in Language Models for Code Intelligence: A Taxonomy and Survey. *arXiv preprint arXiv:2310.17903* (2023). https://arxiv.org/abs/2310.17903

[19] Phannachitta Thongtanunam, David Lo, Baishakhi Ray, and Chris Parnin. 2023. AutoTransform: Learning to Recommend Code Changes Based on Historical Edits. In *Proceedings of the 45th International Conference on Software Engineering (ICSE).* IEEE/ACM, 285–296. https://doi.org/10.1109/ICSE48619.2023.00032

[20] Cybersecurity Ventures. 2020. Cybercrime To Cost The World $10.5 Trillion Annually By 2025. https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/. Online Article.

[21] Yue Wang, Pengcheng Yin, Graham Neubig, Hung-Yu Chan, Shih-Hsiang Lin, Zi-Yi Liu, Chien-Sheng Chang, Xiang Chen, Bill Yuchen Lin, and Dragomir Radev. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Association for Computational Linguistics, 8696–8708. https://arxiv.org/abs/2109.00859