VulRepair: A T5-Based Automated Software Vulnerability Repair

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, Dinh Phung

Problem

- Adversaries take advantage of software vulnerabilities
- According to the National Vulnerability Database, vulnerabilities increased from **4k+/year in 2011 to 20k+/year in 2021**
- Security-analysts are under-resourced when it comes to finding, detecting, and localizing vulnerabilities,
    - But lots of previous work in using AI to predict vulnerabilities
- Even with such tools, security-analysts **spend considerable manual effort** repairing vulnerable functions

Background

- Automated Vulnerability Repair can be formulated as a Neural Machine Translation (NMT) Task
  - Learns mapping between vulnerable function and the repaired function
- Uses Encoder-Decoder layers where the Encoder takes a sequences of vulnerable function tokens and maps it to an intermediate hidden state H
- Decoder takes H and generates output sequence of repair tokens
- Uses following equation to maximize the conditional probability:

$$p(Yi \mid Xi) = p(y_1, ..., y_m \mid x_1, ..., x_n) = \prod i{=}1,m\ p(y_i \mid H, y_1, ..., y_{i-1})$$

- Recurrent Neural Networks (RNNs) were used as NMT models for software engineering tasks, but have subpar performance as they **forget past information for a long sequence of tokens (common in source code)**
- This makes Transformer-based NMT models better as they **do not process tokens sequentially** (they have a context vector for any position within the input vector via the self attention mechanism)

Previous Work: VRepair

- Chen et al. proposed **VRepair**, which uses the Transformer-based NMT architecture for vulnerability repair
- VRepair tokenizes vulnerable input functions by using a **word-level Clang tokenizer with a copy mechanism**
- Code representation fed into the Encoder-Decoder Transformer
- Uses beam search to generate 50 vulnerable repair candidates

- However, VRepair has limitations:
  - **1) Trained on a small bug-fix corpus**
  - 2) Uses word-level tokenization and copy mechanism to handle the Out-Of-Vocabulary (OOV) problem
    - **Cannot generate new tokens that never appear in the input sequence but are newly introduced in the output sequence**
  - 3) Uses Vanilla Transformer's absolute positional encoding
    - **Limits ability of self-attention mechanism to learn relative position of code tokens**
    - Can pay attention to incorrect tokens, such as parentheses instead of variables

This Paper: VulRepair

- Authors of this paper propose VulRepair:
  - Uses a **pre-trained CodeT5** component from a large codebase, CodeSearchNet+C/C#
  - Employs **Byte Pair Encoding (BPE)** to perform subword level tokenization to handle the Out-Of-Vocabulary (OOV) problem
    - **BPE splits rare tokens into meaningful subwords and preserves common tokens**
  - Uses T5 architecture that **considers relative positional information in the self-attention mechanism**

- 12 Encoder layers, 12 Decoder layers, Linear and Softmax Layer
- Scaled dot-product self-attention with **relative positional encoding**
- The relative positional information, P, is supplied to the model as an additional component to K and V

$$Attention(Q, K, V) = \text{softmax}(Q (K+P) T \sqrt{dk}) (V + P)$$

- Self-attention mechanism has **multiple heads**
- Uses **beam search** to generate candidate repairs

Experimental Design

- Authors use the **CVE Fixes** and the **BigVul** datasets for their experiments
- Pre-processed data so vulnerable function contains CWE type, and vulnerable function and repair span have tags
    - Authors ensure special tags would not get treated as code tokens and model could focus on the vulnerability and repair
- Split dataset into **70% training, 10% validation, and 20% testing**
- Used the CodeT5 tokenizer and pre-trained model (12 Transformer Encoder blocks, 12 Transformer Decoder blocks, 768 hidden sizes, and 12 attention heads)
- Used the following cross-entropy loss function to update the model:

$$(H(p, q) = -\sum_{x \in X} p(x) \log q(x))$$

- Used **validation set to fine-tune** the weights

RQ1 Results

What is the accuracy of VulRepair for generating software vulnerability repairs?

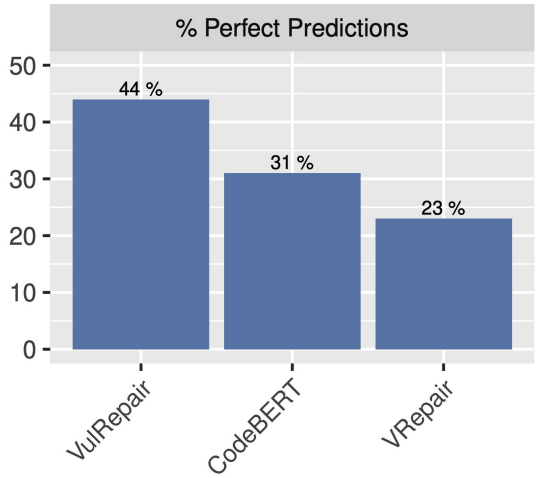

Figure 2: (RQ1) The experimental results of our VulRepair and the two baseline comparisons for vulnerability repairs. (↗) Higher % Perfect Predictions = Better.

RQ2 Results

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

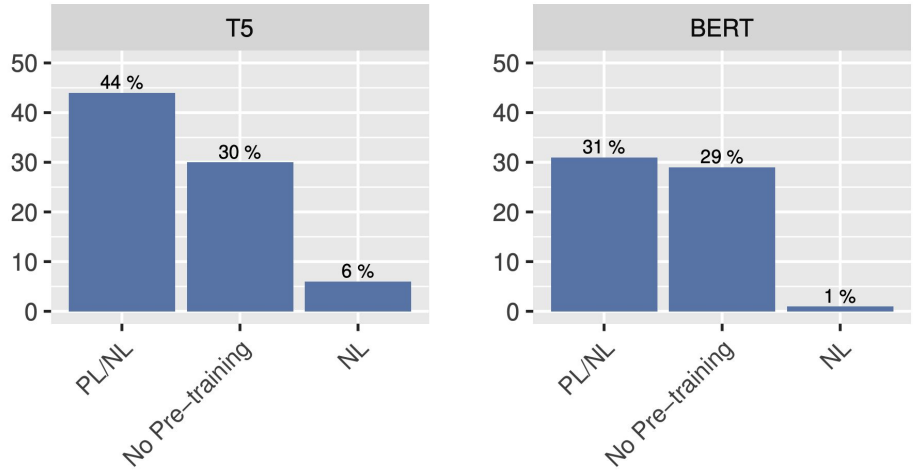

Figure 3: (RQ2) The experimental results of the ablation study with six different models. (↗) Higher % Perfect Predictions = Better.

RQ3 Results

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

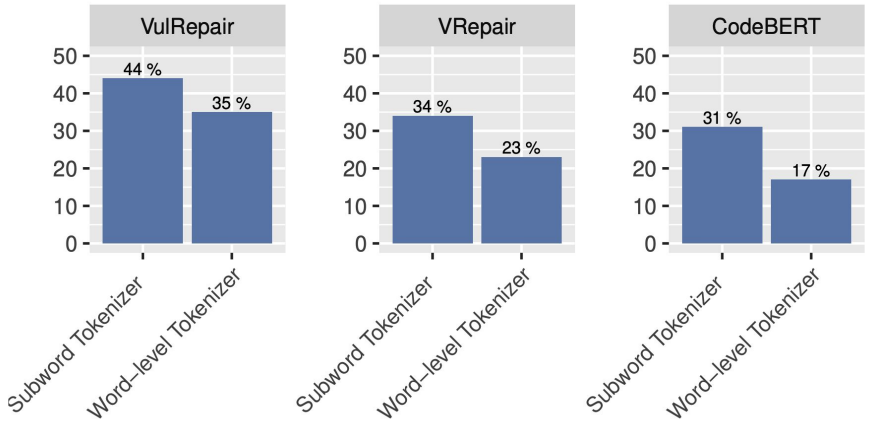

Figure 4: (RQ3) The experimental results of various approaches with different tokenization techniques for vulnerability repairs. (↗) Higher %Perfect Predictions = Better.

RQ4 Results

What are the contributions of the components of VulRepair?

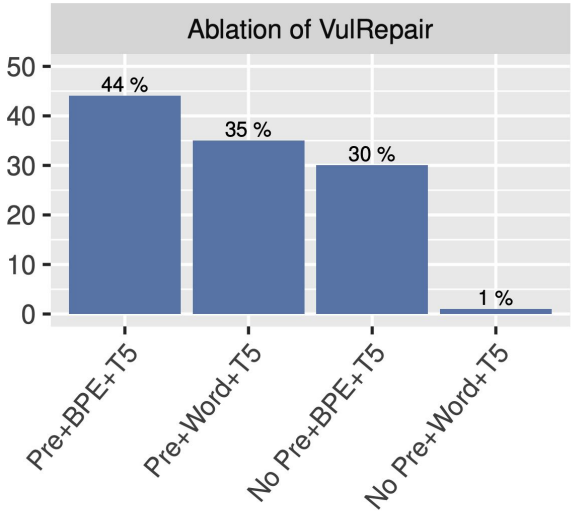

Figure 5: (RQ4) The ablation study result of VulRepair. (↗) Higher %Perfect Predictions = Better.

Types of CWEs VulRepair Can Correctly Repair

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

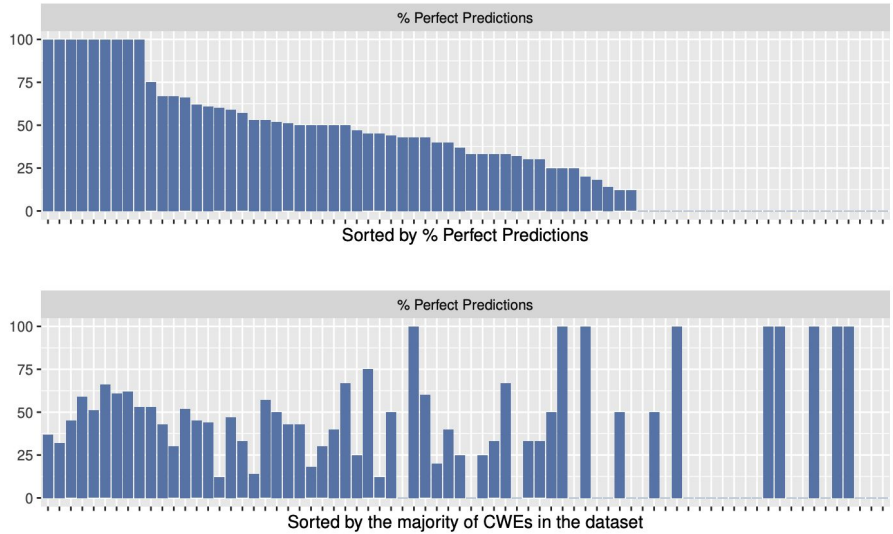

Figure 6: (Discussion) The %Perfect Predictions (y-axis) of our VulRepair according to each type of CWE (x-axis, sorted by % perfect predictions and sorted by the majority of CWEs in the dataset). Detailed statistics can be found in Appendix.

Impact of Function Lengths and Repair Lengths

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%

Impact of the Complexity of the Input

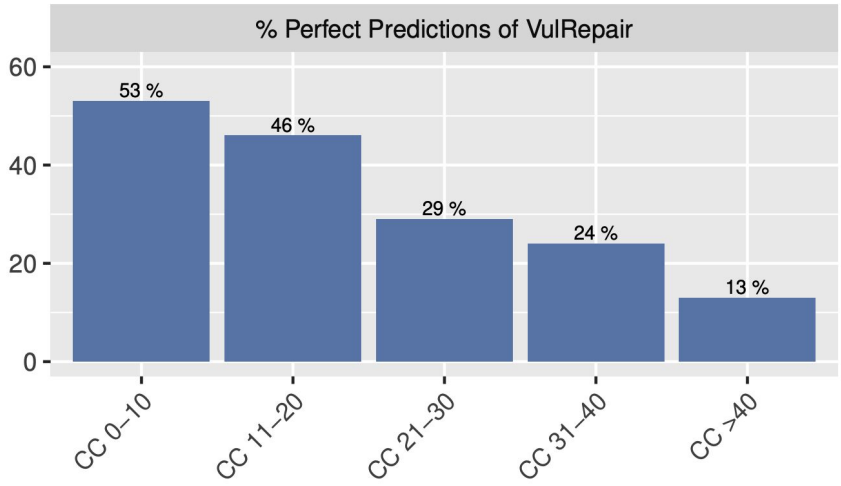

Figure 7: (Discussion) The accuracy of our VulRepair for various ranges of the Cyclomatic Complexity of the input vulnerable functions in the testing set. (↗) Higher % Perfect Predictions = Better.

How Well Does VulRepair Handle the OOV Problem?

- Among 1,706 pairs in the testing dataset, 627 pairs had new tokens in the vulnerable repair
- Among the 627 pairs, VulRepair was able to correctly repair 37% of them, or 234 functions
- **VRepair cannot correctly repair any of these 627 pairs** since it uses the copy mechanism
- However VulRepair's **correct** vulnerability repairs have **1-12 new tokens** while the **incorrect** vulnerability repairs have **1-100 new tokens**

Impact of Beam Size on VulRepair

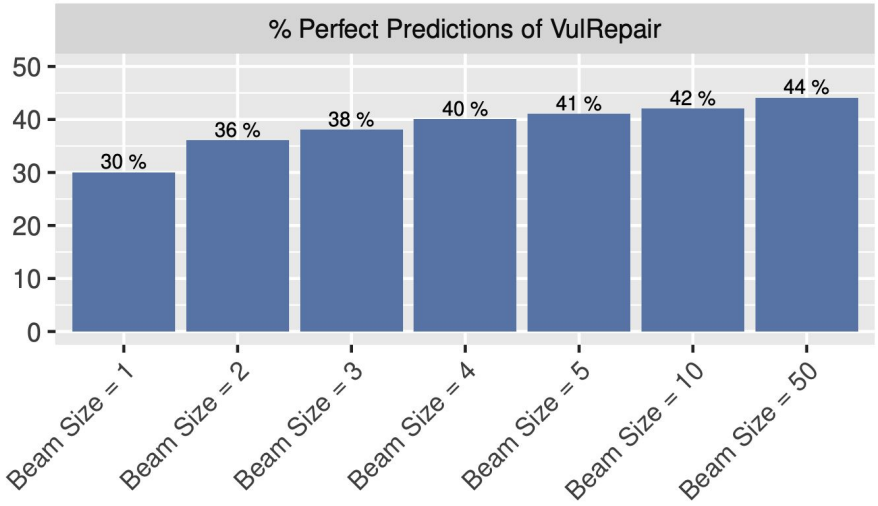

Figure 8: (Discussion) The performance of our VulRepair with different values of beam size. (↗) Higher % Perfect Predictions = Better.

Threats to Validity

- VulRepair only evaluated on the CVEFixes and BigFul datasets → results **may not generalize** to other datasets
- All results are a **lower bound**
  - Authors **did not conduct any hyperparameter tuning** because of large search space
  - Evaluated using % perfect prediction metric → but **models may give correct repairs that don’t match ground-truth data**

My Thoughts

- Not much work within this space using Transformer-based NMT models, so good continuation of previous work, VRepair
- However, more work needs to be done to handle complex repairs + functions
- % perfect prediction metric definitely hurt the performance of the model as reported
- We need a vulnerability dataset that contains unit test cases for code generation tasks in general, including vulnerable repairs
    - The need has shown up in a couple of papers so far