Code Security Vulnerability Repair Using Reinforcement Learning with Large Language Models

Nafis Tanveer Islam, Peyman Najafirad

Department of Computer Science
University of Texas at San Antonio

Abstract

With the recent advancement of Large Language Models (LLMs), generating functionally correct code has become less complicated for a wide array of developers. While using LLMs has sped up the functional development process, it poses a heavy risk to code security. Code generation with proper security measures using LLM is a significantly more challenging task than functional code generation. Security measures may include adding a pair of lines of code with the original code, consisting of null pointer checking or prepared statements for SQL injection prevention. Currently, available code repair LLMs generate code repair by supervised fine-tuning, where the model looks at cross-entropy loss. However, the original and repaired codes are mostly similar in functionality and syntactically, except for a few (1-2) lines, which act as security measures. This imbalance between the lines needed for security measures and the functional code enforces the supervised fine-tuned model to prioritize generating functional code without adding proper security measures, which also benefits the model by resulting in minimal loss. Therefore, in this work, for security hardening and strengthening of generated code from LLMs, we propose a reinforcement learning-based method for program-specific repair with the combination of semantic and syntactic reward mechanisms that focus heavily on adding security and functional measures in the code, respectively.

Introduction

A software vulnerability is a possible set of flaws or weaknesses in the system that allows an attacker to gain access to the system, halt the service provided by the system, or ask for ransom from the software vendors. This becomes even more challenging when threat actors use malicious techniques to steal confidential information or cause economic damage (Dowd, McDonald, and Schuh 2006). In an era of escalating cyber threats, the Open Source Software (OSS) community stands at an unprecedented crossroads, safeguarding the foundation of countless software systems and products integral to modern society. Nevertheless, security vulnerabilities in poorly written software play a vital role across government and its infrastructure, causing widespread disruption of services (Synopsys 2023).

The wide use of large language models further enhances the security vulnerability. Their functional correctness is heavily challenged by the potential introduction of security issues within the generated code (Sandoval et al. 2023; Pearce et al. 2022b). Despite the democratization of coding, providing increased accessibility and productivity among developers from code generated by large language models frequently falls short of established software security standards, possibly harboring vulnerabilities in approximately 40% of cases (Pearce et al. 2022a). This security challenge extends beyond individual models, with various evaluations revealing that other cutting-edge LMs (Nijkamp et al. 2023b; Fried et al. 2022), similar to Copilot, display identical security issues, as highlighted in (Li et al. 2023). A separate investigation (Khoury et al. 2023) identified that ChatGPT, in 16 out of 21 security-relevant cases, generates code falling below minimal security standards. The more alarming concern is the training data of generative AI models, deeply rooted in the same insecure public repositories (Chen et al. 2021). As a result, unless pre-trained explicitly in security, generative AI models give rise to a vicious cycle of generating functional code by disseminating vulnerability.

The pre-training of large language models generally follows the pre-training procedure for generating texts. CodeT5 (Wang et al. 2021) used three pre-training methods to pre-train the model on code. Initially, they tasked the model with predicting a masked set of tokens of code; then, they tasked the model with tagging the identifiers of the code tokens and, finally, predicting the identifiers. StarCoder (Li et al. 2023) pre-trains their model by training it with masked language modeling and next sequence prediction. For functional code generation of LLMs, CodeRL (Le et al. 2022) fine-tunes the model to degenerate programs based on feedback from example unit tests and critic scores.

Furthermore, functional code generation is a task where the model has to generate entire code based on specific instructions. However, Code1 and Code2 in Figure 1 demonstrate that security repair merely adds a few lines to the code. Here, Code 1 is the ground truth repaired function, and Code 2 is the model-generated outcome that attempts to repair a vulnerable function. Original input to the model is the exclusion of the line `dest[99] = NULL`. From the ground truth and the generated repair by the model, we observe that the cross-entropy (CE) loss between the original

```
// Code 1: Vulnerable Code
static void Buffer_Overread (src, dest){
    ...
    for(i=0; i < 99; i++) {
        dest[i] = src[i];
    }
}
```

```
// Code 2: Repaired Code
static void Buffer_Overread_Repair(src, dest) {
   ...
     for(i=0; i < 99; i++){
         dest[i] = src[i];
     }
     dest[99] = NULL;
}
```

CE(Code1, Code2) = 0.005

BERT_Score(Code1, Code2) = 0.63

CodeBLEU(`dest[99] = NULL;` , `dest[99] = "N";`) = 0.13

Figure 1: An illustrative example showcases how the Cross-Entropy (CE) emphasizes only functionality and neglects security. Furthermore, we demonstrate that the BERT_Score and CodeBLEU scores show an improvement in comparing functionality and security measures.

and model outcomes is very low. Thus, the model is tempted to ignore the security measures and generate code almost similar to the input since it yields minimum loss. Moreover, BERTScore on Code1 and Code2 shows higher loss values. Similarly, if we only compare the security measure with a syntactic similarity-based measurement CodeBLEU, we still see that the loss value remains much higher. These higher values indicate that cross-entropy loss is insufficient to make the generative model security aware.

To address these challenges of generating secure code while maintaining functionality, we used a reinforcement learning (RL) based technique for vulnerability repair of the source with a specialized reward function designed specifically for enhancing code security. To address the issue of security-focused comparison of the generated code and ground truth, we propose a syntactic reward mechanism that rewards the RL model only when proper security measures are added to the code. Furthermore, to ensure the functional correctness of our proposed system, we propose a syntactic and semantic reward function. Therefore, we propose *SecureCode* powered by a large language model CodeGen2 (Nijkamp et al. 2023a) trained using reinforcement learning with the combination of syntactic and semantic reward to generate security repairs of vulnerable code that are functionally correct.

In summary, the contributions of this paper are:

- We introduce a state-of-the-art LLM-powered tool to repair code vulnerabilities written in C/C++. Our proposed system leverages reinforcement learning to generate secure code while maintaining functionality.
- We introduce the combination of syntactic and semantic reward values to enhance the capability of our proposed reinforcement learning model to generate secure and functional code.
- Our research also includes a quantitative analysis of the results to demonstrate the security repair quality of our proposed model.

Related Works

Code vulnerabilities have profound implications across diverse domains in the digital realm, ranging from the utilization of digital devices within IoT ecosystems and online accounts (Atashpanjeh et al. 2022) to pivotal systems like containers (Haq, Tosun, and Korkmaz 2022) and operating systems. Although anticipating specific sophisticated techniques proves challenging, most of these vulnerabilities can be attributed to developers' setbacks in ensuring robust code security. This intersection of AI and cybersecurity emphasizes the critical role of proactive measures in fortifying our digital infrastructure against potential threats.

Code Vulnerability with LLMs: Most Large Language Models (LLMs) exhibit minimal concern for security aspects in programming languages. Pearce (Pearce et al. 2022b) scrutinized LLMs' zero-shot vulnerability repair capabilities and the challenges posed by bugs. Experimental findings indicated that while LLMs can generate bug fixes, they require a specifically crafted prompt for addressing a particular bug. SVEN (He and Vechev 2023) introduced an adversarial technique to assess LLMs, proposing generating safer code by utilizing property-specific continuous vectors to guide program generation. Additionally, studies by (Pearce et al. 2022a), (Jesse et al. 2023), and (Sandoval et al. 2022) extensively explored autocompletion effectiveness, integrating LLMs with various IDEs and analyzing outcomes. While (Pearce et al. 2022a) and (Jesse et al. 2023) concluded that LLMs produce vulnerable code, Sandoval (Sandoval et al. 2022) suggested that LLMs contribute to generating more functional code with improved security features.

Vulnerability Repair: Repairing programs pose a formidable challenge, requiring the identification of vulnerable lines and the subsequent generation of a suitable compilable line for fixing the vulnerability. Earlier approaches, such as those incorporating human-specified safety properties (Huang et al. 2019), enforced constraints like preventing access to memory beyond program boundaries. Zhang et al. (Zhang et al. 2022) introduced methods to identify patch invariants or vulnerable code locations, employing a set of templates for repair generation. More recent approaches like Vrepair (Chen, Kommrusch, and Monperrus 2022) adopt a transformer-based transfer learning strategy to address vulnerabilities in real-world programs. Similarly, VulRepair (Fu et al. 2022) presents a vulnerability repair technique utilizing the pre-trained CodeT5 (Wang et al. 2021) and BPE tokenizer. However, with the emergence of code-based Large Language Models (LLMs) like Codex (Chen et al. 2021), works by Pearce et al. (Pearce et al. 2022b), Jesse et al. (Jesse et al. 2023), and Prenner (Prenner and Robbes 2021) demonstrate some capability of these models to repair vulnerable code through zero-shot learning.

Fine-Tuning of LLMs with RL: Despite the simplicity of fine-tuning a neural network using Supervised Fine-Tuning, recent advancements in Large Language Model training (Touvron et al. 2023; Ouyang et al. 2022) show that although Reinforcement Learning is less intuitive for the human mind, it is highly effective in guiding Large Language Models to follow specific requirements, in our case, security of the code. Additionally, Reinforcement Learning provides a framework to use arbitrary weak signals to align the model. These signals include human preference and quantitative evaluation scores such as BLEU (Papineni et al. 2002), which may not be possible or computationally expensive to utilize in a supervised learning setting due to their non-differentiable nature (Liu 2019). One way to overcome this limitation for Supervised Fine-Tuning is to use differentiable loss functions such as the Cross-Entropy Loss. However, our initial analysis in Figure 1 shows that Cross-Entropy Loss does not perform well in a code security measurement setting. Also, Since SFT signals are strong and explicit, a model fine-tuned using supervised settings may be more prone to losing generality (Lin et al. 2023). Aggregating these reasons motivates more research on using reinforcement learning to generate secure code.

Methodology

One of the major challenges in code generative models is the long-range dependency of source code (Islam et al. 2023), where the declaration and the actual usage of a variable are far apart. Because of the long-range dependency, it is challenging for an LLM to determine the position where to add security measures for a given code. Therefore, we use a causal-decoder model, CodeGen2, that emphasizes the importance of previous tokens for our code repair work with enhanced generalization capability (Wang et al. 2022). In code, the context of a variable or function call might span several lines or even pages; the ability to capture these dependencies is critical for accurate code generation or repair, which a causal decoder model can use effectively and efficiently. Regular encoder-decoder-based LLM architecture shows a bidirectional property where they read tokens from forward and backward. Such a bidirectional architecture causes unnecessary computation that is not needed for code vulnerability repair since the reasoning of a vulnerability always exists before the code breaks or has runtime issues. Moreover, causal decoder models remove the encoder layer. Therefore, they have faster inference time with less memory footprint. Figure 2 demonstrates the overall architecture of our proposed system.

Initially, we convert the vulnerable input function f_{vul} and the ground truth repaired output function f_{rep} into a sequence of tokens, $t_1, t_2, ...t_p \in T$ and $y_1, y_2, ...y_q \in Y$. Then we combine both into a unique sequence $w_1, w_2, ...w_{p+q} = (t_1, ..., t_p, \$, y_1, ..., y_q)$, where the \$ is a special token separating the vulnerable and the repaired program. p is the number of input tokens, and q is the number of output tokens.

Reinforcement Learning for Functional Repair

Given the causal decoder architecture of `SecureCode`, our model is forced to predict the next code token, and the model cannot overlook future tokens by looking at the next token during output generation. The model is provided with the input sequence $t_i \in T$ during inference, which autoregressively generates the output, y_i.

Therefore, we fine-tune our model with reinforcement learning for a code-to-code generation task where the generated code is the repaired version of the original input code. To achieve functional code repair, our proposed reinforcement learning technique combines syntactically and semantically aware reward functions with the Proximal Policy Optimization (PPO) algorithm proposed by Schulman et al. (Schulman et al. 2017). We denote the input function or the vulnerable code as f_{vul}. The repaired output generated the model as $\hat{f_{rep}}$, and the ground truth repaired function is f_{rep}. We assume the output sequence is $w_1^r, w_2^r, ...w_k^r$, where k is the total number of output tokens of the generated repair.

We define the token-wise code generative process as a deterministic Contextual Markov Decision Process (Hallak, Di Castro, and Mannor 2015) with observable context only from previous tokens of vulnerable code. The repaired code sequence generated, which is the state at the k^{th} token generation, is defined by our policy $\pi(.|w^r : k-1, t)$, which is the probability distribution of the previous $k - 1^{th}$ input tokens from vulnerable function f_{vul}.

Policy Optimization: The reinforcement learning objective is to find the optimal policy by maximizing the cumulative syntactic and semantic reward by adding security measures while keeping code functionality checked.

To meet this requirement, we introduce a syntactic code evaluation technique, CodeBLEU (Ren et al. 2020), as a reward value to check the syntactic similarity between the repaired lines generated by the model and the ground truth repair. Here in Figure 2, the repaired line refers to the new line the model is expected to generate to add security measures to the code. Furthermore, to ensure the functional correctness in the generated outcome, we utilize BERTScore (Zhang et al. 2019) as a semantic reward value that quantifies the semantic similarity or functionality between the entire input vulnerable code and the generated repaired code. If the original vulnerable code is f_{vul}, the repaired code generated by the model is $\hat{f_{rep}}$, and the ground truth repaired

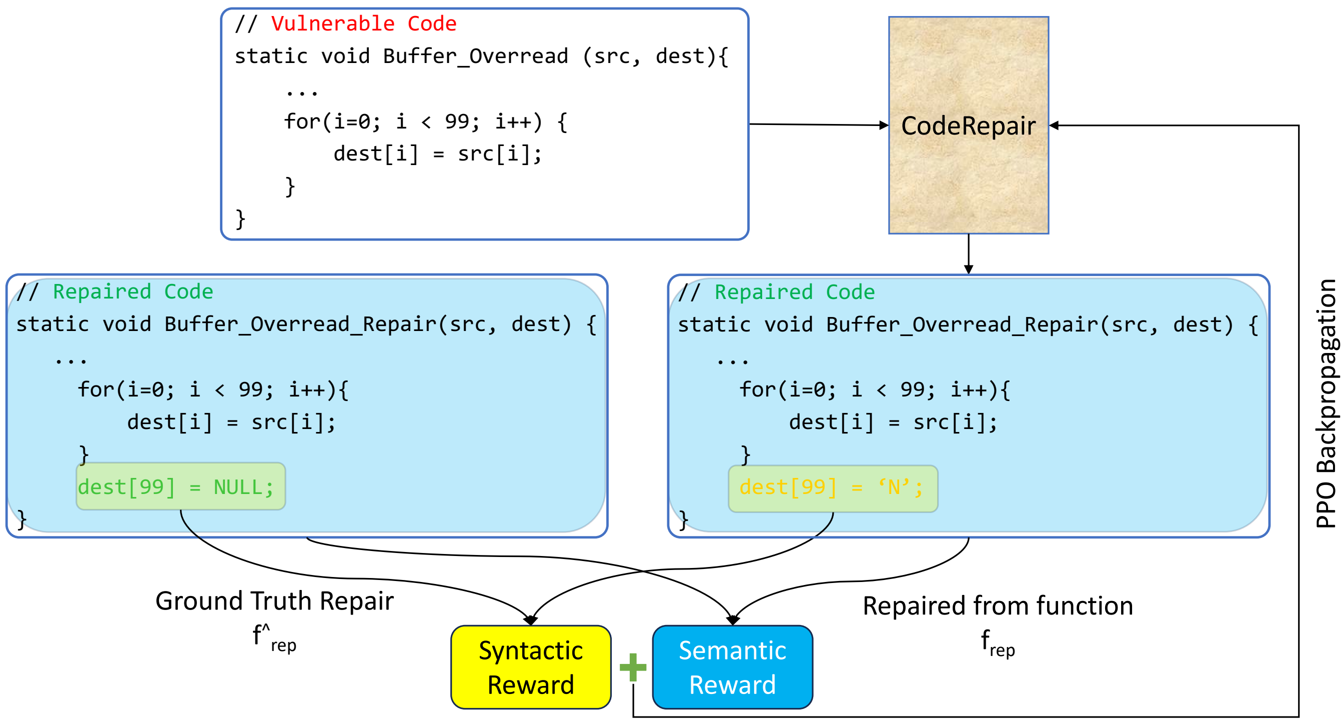

Figure 2: A high-level overview of our proposed CodeRepair System with Reinforcement learning with semantic and syntactic loss.

code is f_{rep}. As such, we calculate the policy optimization r_θ, using the following equation:

$$L(r_\theta) = log(\sigma(r_\theta(f_{vul}, f_{rep}) - r_\theta(f_{vul}, \hat{f_{rep}})) \quad (1)$$

where $r_\theta(f_{vul}, f_{rep})$ and $r_\theta(f_{vul}, \hat{f_{rep}})$ is the scalar output of the reward model for the vulnerable code f_{vul}. Here σ is an activation function, and θ is a learnable parameter.

Reward: We calculate the reward by combining the CodeBLEU (Ren et al. 2020) score and BERTScore (Zhang et al. 2019). CodeBLEU is the weighted combination BLEU score (Papineni et al. 2002), $BLEU_{weight}$ is the weighted n-gram match, obtained by comparing the generated code and the ground truth repaired code tokens, $Match_{ast}$ is the AST match, exploring the syntactic information of code, and $Match_{df}$ is the dataflow match, considering the similarity between ground truth and generate code.

$$R_{CodeB} = \alpha.B + \beta.B_{weight} + \delta.Match_{ast} + \gamma.Match_{df} \quad (2)$$

Here, B in the equation stands for BLEU.

Furthermore, we use BERTScore for semantic comparison using cosine similarity score. A BERT vector represents tokens that permit the generation of a soft similarity measure instead of exact matching. The cosine similarity of a reference token from ground truth repair t_r^i and a candidate token $\hat{w_r^i}$, we calculate the cosine similarity as $(t_r^i)^T \hat{w_r^i}$. Therefore, the F1 measurement of the BERTScore stands as follows;

$$R_{BERT} = \frac{1}{|t_r^i|} \sum_{t_r^i \in T} max(t_r^i)^T \hat{w_r^i} \quad (3)$$

where R_{BERT} is our expected BERTScore. We combine Equation 2 and 3 to get the final reward value. The final reward is calculated as follows:

$$R = R_{CodeB} + R_{BERT} \quad (4)$$

Here, R is the final calculated reward value.

Experiments

Dataset

We use the VulDeeLocator dataset (Li et al. 2021) for C/C++ programs with vulnerable source code and their corresponding repairs. The code is sourced from the National Vulnerability Database (NVD) and the Software Assurance Reference Dataset (SARD). This dataset has 40,382 vulnerable and 115,157 not-vulnerable code snippets.

Evaluation Metrics

BLEU Score: The BLEU (Papineni et al. 2002) score evaluates machine-generated text where the score ranges between 0 and 1. A value of 0 means that the generated output does not have a single n-gram match with the original output, while 1 means a perfect n-gram match with the original outcome. However, it's important to note that BLEU is a reference-based metric and may not capture all aspects of translation quality, such as fluency or semantic accuracy.

Model	Parameter	BLEU	Rouge-L
CodeGen2 (SFT)	1B	0.50	0.56
	3.7B	0.70	0.69
	7B	0.80	0.81
CodeRepair (RL)	1B	0.70	0.73
	3.7B	0.76	0.80
	7B	**0.86**	**0.88**

Table 1: BLEU, Rouge-L Evaluation on generating code repairs from vulnerable code inputs.

The final BLEU score is calculated using sentence-level n-gram precision and the brevity penalty. The n-gram precision is weighted based on the length of the n-grams (from unigrams to a specified maximum n-gram order, typically 4). The geometric mean of the weighted n-gram precisions is multiplied by the brevity penalty to obtain the BLEU score.

Rouge-L: Similar to BLEU, Rouge-L (Lin 2004) score is also a number between 0 and 1 to measure the similarity of two generated texts. It generates a score by quantifying precision and recall by examining the longest common subsequence (LCS) between the generated and reference codes. Precision, in this context, refers to the ratio between the length of the LCS and the length of the generated code, whereas recall refers to the ratio between the length of the LCS and the length of the reference code. These metrics reveal the degree of consistency between the generated code and the reference code, highlighting their similarity and concordance. ROUGE-L is a valuable metric for evaluating the effectiveness of code vulnerability analysis algorithms by quantifying the quality of the generated code relative to the reference code.

Experimental Results and Discussion

In our experiments, we split datasets randomly into 80:10:10 for training, validation, and testing. We utilized a pretrained CodeGen2 model with 32 decoder layers. We trained for 20 epochs with a max token length of 512. Our 7B parameter model uses a learning rate of $2e^{-5}$, a batch size of 2, a beam size of 4, and a temperature value of 0.5 for optimal performance. The training was conducted on 8 NVIDIA A100 GPUs with 40 GB of memory each. Integration of DeepSpeed (Rajbhandari et al. 2020) with our HuggingFace CodeGen2 implementation for minimized memory footprints.

To ensure our proposed system is repairing vulnerability accurately and reliably, we compare our proposed method with the regular fine-tuning-based method. We did two types of training to generate repairs of the vulnerable code. We used 1B, 3.7B, and 7B variants of the CodeGen2 model for comparison. We trained these three variants using Supervised Fine Tuning (SFT) and our proposed Reinforcement Learning (RL) technique for comparative analysis. In order to quantitatively evaluate the repairs, we use BLEU and Rouge-L metrics.

Discussion: Our empirical results highlight the superior performance of our model when compared to counterparts with comparable or fewer parameters. As outlined in Table 1, our model exhibits a notable improvement, surpassing the 7B variant of CodeGen2 (SFT) by 0.06 BLEU and 0.07 Rouge-L score. Particularly noteworthy is the efficacy of our proposed Reinforcement Learning (RL) training approach, outperforming the SFT method across the other two variants. A consistent trend emerges in the consistently higher Rouge-L scores, underscoring its robustness in evaluating similarity based on the longest common subsequence (LCS) between original and generated tokens, as opposed to BLEU's n-gram-centric methodology.

```
void RecordMutation(TF_Graph* graph, const TF_Operation& op,
                    const char* mutation_type) {
  // If any session has already run this node_id, mark this session as
  // unrunnable.
  for (auto it : graph->sessions) {
    {
      mutex_lock session_lock(it.first->mu);
      if (it.first->last_num_graph_nodes > op.node.id()) {
        it.second = strings::StrCat(
            "Operation '", op.node.DebugString(), "' was changed by ",
            mutation_type,
            " after it was run by a session. This mutation will have no effect, "
            );
      }
    }
    // Releasing Lock -- Repaired Line
    mutex_unlock session_unlock(it.first->mu);
  }
}
```

Listing 1: Case Study 1: Vulnerability Repair in Mutex Condition of a Function

Case Study of the Generated Outcomes

Case Study 1 In this section, we will analyze the qualitative outcome between the SFT- and RL-based fine-tuning models. The initial implementation of the complex algorithm exhibited a vulnerability prone to integer overflow. This flaw, residing in accumulating values within the loop, could result in unexpected behavior and potential instability when processing large datasets.

The provided code snippet in Listing 1 is from the TensorFlow repository, where the code handles mutations in a TensorFlow (TF) graph during a session. This sample code is a part of our evaluation dataset. In this function, the code aims to handle mutations to a TensorFlow graph in a concurrent environment, marking sessions as un-runnable if they have already executed a specified operation. Here, the original vulnerable code excludes lines number 16 and 17. This function aims to run exclusively and exit when the loop ends. However, when the function exits in the original function, the mutex is still unlocked, blocking any execution of this function and making this function vulnerable.

A simple solution to avoid this race condition problem is to remove line number 7. However, this way of generating the solution decreases the functionality of the given function. However, when we fine-tune the model using reinforcement learning, the functional and security reward function rewards the model, compared to the cross-entropy loss, which only looks at the tokens holistically without specifically for functionality and security issues. Therefore when the model is trained with reinforcement learning, it retains the functionality by not removing line 7. Moreover, the model added a line at line number 17, where it added a statement where the mutex is unlocked, thereby effectively allowing the next call to execute and repair the vulnerability of the function.

```
1
2
3  static void copyIPv6IfDifferent(void *
       dest, const    void * src)
4
5  {
6
7  if(dest != src && src != NULL) {
8
9    memcpy(dest, src, sizeof(struct
                 in6_addr));
10
11   }
```

Listing 2: Case Study2: Buffer Overflow Vulnerability Repair

Case Study 2: The original implementation of the copyIPv6IfDifferent in Listing 2 function demonstrates a vulnerability where the function is designed to copy an IPv6 address from the source to the destination only if they are different. However, the vulnerability lies in the reliance on a basic memory copy operation without verifying the size of the source and destination data. In this case the the input code from Listing 2 without line number 7.

We introduced a comprehensive check to address this vulnerability before initiating the memory copy operation. The enhanced version now not only ensures the inequality of source and destination addresses but also validates the memory regions to avoid potential buffer overflows. This modification safeguards against unintended modifications to the destination data and reinforces the reliability of the `copyIPv6IfDifferent` function.

This analysis shows a vulnerability within the `copyIPv6IfDifferent` function. While the objective is straightforward, the original code lacked checks to ensure the integrity of the source and destination data, opening avenues for potential memory-related vulnerabilities. Our revised implementation introduces a layered security approach, incorporating checks for both address inequality and memory region validation. Our RL model added line 7 to repair the vulnerability, while the SFT model generates the exact same code as the input.

```
1  static memcached_return_t
     _read_one_response(
     memcached_instance_st *instance, char
      *buffer,
2
          const size_t buffer_length,
3
          memcached_result_st *result) {
4    memcached_server_response_decrement(
       instance);
5    if (result == NULL) {
6      Memcached *root = (Memcached *)
       instance->root;
7      result = &root->result;
8    }
9    memcached_return_t rc;
10   if (memcached_is_binary(instance->root
       )) {
11     do {
12       rc = binary_read_one_response(
       instance, buffer, buffer_length,
       result);
13     } while (rc ==
       MEMCACHED_FETCH_NOTFINISHED);
14   } else {
15     rc = textual_read_one_response(
       instance, buffer, buffer_length,
       result);
16   }
17
18  -if (memcached_fatal(rc) && rc !=
       MEMCACHED_TIMEOUT) {
19  - memcached_io_reset(instance);
20   }
```

Listing 3: Case Study 3: I/O Operations Vulnerability

Case Study 3: The code snippet in Listing 3 is a vulnerable function where in the conditional statement $if(memcached_fatal(rc)$ && $rc! = MEMCACHED_TIMEOUT)$. This code segment is susceptible to logical errors that could lead to unexpected behavior. Specifically, the intention will reset the I/O operations $(memcached_io_reset(instance))$ when the return code (rc) indicates a fatal error that is not a timeout. If the condition is met, the following code block, intended for error handling, may not execute as expected, leaving the system in an inconsistent state.

To address this vulnerability, the solution is to remove this line so that the inconsistency does not happen. However, the repair is relatively difficult to understand because the model is not given any other function for external analysis. Therefore, the SFT and RL models both removed lines 18 and 19 and repaired the vulnerability.

From the analysis, we see that our model is more efficient and effective when additional lines must be added to repair the vulnerability. Our proposed RL model learns to add lines that add security measures to the code since the model was rewarded for keeping the functionality and security aspects intact. However, both models perform similarly when repairing the vulnerability, including removing some lines.

Conclusion

Our study presents a solution to the currently existing code repair generation challenges using large language models. We proposed *CodeRepair* with a reinforcement learning technique that ensures the functional and security aspects of code repair generation. In order to ensure functional and semantic correctness, we propose to use the combination of a syntactic and semantic reward called CodeBLEU and BERTScore. Our experimental results show that our model performed superior compared to previous code repair training methods like supervised fine-tuning. Furthermore, we demonstrated three case studies with real-world vulnerable code which is a part of our evaluation dataset. Our case study demonstrates that our RL model is more effective in repairing vulnerabilities when a security measure needs to be added to the code to generate the repair.

References

Atashpanjeh, H.; Behfar, A.; Haverkamp, C.; Verdoes, M. M.; and Al-Ameen, M. N. 2022. Intermediate Help with Using Digital Devices and Online Accounts: Understanding the Needs, Expectations, and Vulnerabilities of Young Adults. In *International Conference on Human-Computer Interaction*, 3–15. Springer.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, Z.; Kommrusch, S.; and Monperrus, M. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1): 147–165.

Dowd, M.; McDonald, J.; and Schuh, J. 2006. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.-t.; Zettlemoyer, L.; and Lewis, M. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Fu, M.; Tantithamthavorn, C.; Le, T.; Nguyen, V.; and Phung, D. 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*, 935–947.

Hallak, A.; Di Castro, D.; and Mannor, S. 2015. Contextual markov decision processes. *arXiv preprint arXiv:1502.02259*.

Haq, M. S.; Tosun, A. Ş.; and Korkmaz, T. 2022. Security Analysis of Docker Containers for ARM Architecture. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 224–236. IEEE.

He, J.; and Vechev, M. 2023. Controlling large language models to generate secure and vulnerable code. *arXiv preprint arXiv:2302.05319*.

Huang, Z.; Lie, D.; Tan, G.; and Jaeger, T. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*, 539–554. IEEE.

Islam, N.; Parra, G.; Manuel, D.; Bou-Harb, E.; and Najafirad, P. 2023. An Unbiased Transformer Source Code Learning with Semantic Vulnerability Graph. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 144–159. Los Alamitos, CA, USA: IEEE Computer Society.

Jesse, K.; Ahmed, T.; Devanbu, P. T.; and Morgan, E. 2023. Large Language Models and Simple, Stupid Bugs. *arXiv preprint arXiv:2303.11455*.

Khoury, R.; Avila, A. R.; Brunelle, J.; and Camara, B. M. 2023. How Secure is Code Generated by ChatGPT? *arXiv preprint arXiv:2304.09655*.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35: 21314–21328.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; and Jin, H. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*.

Lin, C.-Y. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, 74–81.

Lin, Y.; Tan, L.; Lin, H.; Zheng, Z.; Pi, R.; Zhang, J.; Diao, S.; Wang, H.; Zhao, H.; Yao, Y.; and Zhang, T. 2023. Speciality vs Generality: An Empirical Study on Catastrophic Forgetting in Fine-tuning Foundation Models. arXiv:2309.06256.

Liu, T. 2019. Optimizing BLEU Scores for Improving Text Generation.

Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; and Zhou, Y. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023b. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR*.

Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C. L.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; Schulman, J.; Hilton, J.; Kelton, F.; Miller, L.; Simens, M.; Askell, A.; Welinder, P.; Christiano, P.; Leike, J.; and Lowe, R. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 311–318.

Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2022a. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. IEEE.

Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; and Dolan-Gavitt, B. 2022b. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*, 1–18. IEEE Computer Society.

Prenner, J. A.; and Robbes, R. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922*.

Rajbhandari, S.; Rasley, J.; Ruwase, O.; and He, Y. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16. IEEE.

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Sandoval, G.; Pearce, H.; Nys, T.; Karri, R.; Dolan-Gavitt, B.; and Garg, S. 2022. Security Implications of Large Language Model Code Assistants: A User Study. *arXiv preprint arXiv:2208.09727*.

Sandoval, G.; Pearce, H.; Nys, T.; Karri, R.; Garg, S.; and Dolan-Gavitt, B. 2023. Lost at c: A user study on the security implications of large language model code assistants. *In USENIX Security. arXiv preprint arXiv:2208.09727*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Synopsys. 2023. OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; Bikel, D.; Blecher, L.; Ferrer, C. C.; Chen, M.; Cucurull, G.; Esiobu, D.; Fernandes, J.; Fu, J.; Fu, W.; Fuller, B.; Gao, C.; Goswami, V.; Goyal, N.; Hartshorn, A.; Hosseini, S.; Hou, R.; Inan, H.; Kardas, M.; Kerkez, V.; Khabsa, M.; Kloumann, I.; Korenev, A.; Koura, P. S.; Lachaux, M.-A.; Lavril, T.; Lee, J.; Liskovich, D.; Lu, Y.; Mao, Y.; Martinet, X.; Mihaylov, T.; Mishra, P.; Molybog, I.; Nie, Y.; Poulton, A.; Reizenstein, J.; Rungta, R.; Saladi, K.; Schelten, A.; Silva, R.; Smith, E. M.; Subramanian, R.; Tan, X. E.; Tang, B.; Taylor, R.; Williams, A.; Kuan, J. X.; Xu, P.; Yan, Z.; Zarov, I.; Zhang, Y.; Fan, A.; Kambadur, M.; Narang, S.; Rodriguez, A.; Stojnic, R.; Edunov, S.; and Scialom, T. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

Wang, T.; Roberts, A.; Hesslow, D.; Le Scao, T.; Chung, H. W.; Beltagy, I.; Launay, J.; and Raffel, C. 2022. What language model architecture and pretraining objective works best for zero-shot generalization? In *International Conference on Machine Learning*, 22964–22984. PMLR.

Wang, Y.; Wang, W.; Joty, S.; and Hoi, S. C. 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*, 8696–8708. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics.

Zhang, T.; Kishore, V.; Wu, F.; Weinberger, K. Q.; and Artzi, Y. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Zhang, Y.; Gao, X.; Duck, G. J.; and Roychoudhury, A. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 691–702.

References

Atashpanjeh, H.; Behfar, A.; Haverkamp, C.; Verdoes, M. M.; and Al-Ameen, M. N. 2022. Intermediate Help with Using Digital Devices and Online Accounts: Understanding the Needs, Expectations, and Vulnerabilities of Young Adults. In *International Conference on Human-Computer Interaction*, 3–15. Springer.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chen, Z.; Kommrusch, S.; and Monperrus, M. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1): 147–165.

Dowd, M.; McDonald, J.; and Schuh, J. 2006. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.-t.; Zettlemoyer, L.; and Lewis, M. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Fu, M.; Tantithamthavorn, C.; Le, T.; Nguyen, V.; and Phung, D. 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*, 935–947.

Hallak, A.; Di Castro, D.; and Mannor, S. 2015. Contextual markov decision processes. *arXiv preprint arXiv:1502.02259*.

Haq, M. S.; Tosun, A. Ş.; and Korkmaz, T. 2022. Security Analysis of Docker Containers for ARM Architecture. In *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, 224–236. IEEE.

He, J.; and Vechev, M. 2023. Controlling large language models to generate secure and vulnerable code. *arXiv preprint arXiv:2302.05319*.

Huang, Z.; Lie, D.; Tan, G.; and Jaeger, T. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*, 539–554. IEEE.

Islam, N.; Parra, G.; Manuel, D.; Bou-Harb, E.; and Najafirad, P. 2023. An Unbiased Transformer Source Code Learning with Semantic Vulnerability Graph. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 144–159. Los Alamitos, CA, USA: IEEE Computer Society.

Jesse, K.; Ahmed, T.; Devanbu, P. T.; and Morgan, E. 2023. Large Language Models and Simple, Stupid Bugs. *arXiv preprint arXiv:2303.11455*.

Khoury, R.; Avila, A. R.; Brunelle, J.; and Camara, B. M. 2023. How Secure is Code Generated by ChatGPT? *arXiv preprint arXiv:2304.09655*.

Le, H.; Wang, Y.; Gotmare, A. D.; Savarese, S.; and Hoi, S. C. H. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35: 21314–21328.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; and Jin, H. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*.

Lin, C.-Y. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, 74–81.

Lin, Y.; Tan, L.; Lin, H.; Zheng, Z.; Pi, R.; Zhang, J.; Diao, S.; Wang, H.; Zhao, H.; Yao, Y.; and Zhang, T. 2023. Speciality vs Generality: An Empirical Study on Catastrophic Forgetting in Fine-tuning Foundation Models. arXiv:2309.06256.

Liu, T. 2019. Optimizing BLEU Scores for Improving Text Generation.

Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; and Zhou, Y. 2023a. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023b. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR*.

Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C. L.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; Schulman, J.; Hilton, J.; Kelton, F.; Miller, L.; Simens, M.; Askell, A.; Welinder, P.; Christiano, P.; Leike, J.; and Lowe, R. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 311–318.

Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2022a. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. IEEE.

Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; and Dolan-Gavitt, B. 2022b. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*, 1–18. IEEE Computer Society.

Prenner, J. A.; and Robbes, R. 2021. Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. *arXiv preprint arXiv:2111.03922*.

Rajbhandari, S.; Rasley, J.; Ruwase, O.; and He, Y. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16. IEEE.

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Sandoval, G.; Pearce, H.; Nys, T.; Karri, R.; Dolan-Gavitt, B.; and Garg, S. 2022. Security Implications of Large Language Model Code Assistants: A User Study. *arXiv preprint arXiv:2208.09727*.

Sandoval, G.; Pearce, H.; Nys, T.; Karri, R.; Garg, S.; and Dolan-Gavitt, B. 2023. Lost at c: A user study on the security implications of large language model code assistants. *In USENIX Security. arXiv preprint arXiv:2208.09727*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Synopsys. 2023. OPEN SOURCE SECURITY AND RISK ANALYSIS REPORT.

Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; Bikel, D.; Blecher, L.; Ferrer, C. C.; Chen, M.; Cucurull, G.; Esiobu, D.; Fernandes, J.; Fu, J.; Fu, W.; Fuller, B.; Gao, C.; Goswami, V.; Goyal, N.; Hartshorn, A.; Hosseini, S.; Hou, R.; Inan, H.; Kardas, M.; Kerkez, V.; Khabsa, M.; Kloumann, I.; Korenev, A.; Koura, P. S.; Lachaux, M.-A.; Lavril, T.; Lee, J.; Liskovich, D.; Lu, Y.; Mao, Y.; Martinet, X.; Mihaylov, T.; Mishra, P.; Molybog, I.; Nie, Y.; Poulton, A.; Reizenstein, J.; Rungta, R.; Saladi, K.; Schelten, A.; Silva, R.; Smith, E. M.; Subramanian, R.; Tan, X. E.; Tang, B.; Taylor, R.; Williams, A.; Kuan, J. X.; Xu, P.; Yan, Z.; Zarov, I.; Zhang, Y.; Fan, A.; Kambadur, M.; Narang, S.; Rodriguez, A.; Stojnic, R.; Edunov, S.; and Scialom, T. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288.

Wang, T.; Roberts, A.; Hesslow, D.; Le Scao, T.; Chung, H. W.; Beltagy, I.; Launay, J.; and Raffel, C. 2022. What language model architecture and pretraining objective works best for zero-shot generalization? In *International Conference on Machine Learning*, 22964–22984. PMLR.

Wang, Y.; Wang, W.; Joty, S.; and Hoi, S. C. 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*, 8696–8708. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics.

Zhang, T.; Kishore, V.; Wu, F.; Weinberger, K. Q.; and Artzi, Y. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Zhang, Y.; Gao, X.; Duck, G. J.; and Roychoudhury, A. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 691–702.