LLM Driven Security Vulnerability Repair
Final Report

Vinay Shukla [1] **Lana Lim** [2] **Brian Felipe Brito** [3] **Steven Pan** [4]

Abstract

Security vulnerabilities repair is a difficult task in dire need of automation. Recently, Large Language Models (LLMs) have shown remarkable potential in understanding and interpreting code segments. This has motivated a line of research that applies LLMs to detect and fix vulnerabilities. But currently many of these approaches involve using pretraining procedures to learn task-specific repairs. This project explores the efficacy of employing an out of the box LLM with prompt engineering. We aim to steer a base model's behavior towards the desired outcome without fine-tuning it with current automated program repair techniques (APRs) or adjusting the model's weight. Our approach prompts OpenAI's GPT 3.5 turbo and tests on one primary dataset with 9 different vulnerabilities. Its goal is to generate a "fixed" version of the original code containing a specific vulnerability. Our best prompt can achieve over a 50% repair rate. Furthermore, we propose a novel semi-automatic approach for rating the outputted repairs: LLMs supplemented with human evaluation. This automates the process of evaluation while also maintaining a high-level of confidence in the result.

1. Introduction

In recent years, large language models such as BERT (Devlin et al., 2019), T5 (Raffel et al., 2020), and GPT (Brown et al., 2020) have displayed the ability to perform well on a wide variety of language benchmarks such as summarization, question-and-answering, etc. One such task involves the understanding and behavior of code. Recent models have been tailored to this task such as Codex (Chen et al., 2021). These models are purely data driven and are exposed to a wide variety of text during the training process. This is what ultimately makes them capable of having a high semantic understanding over many textual domains.

Code vulnerabilities have presented itself as an important issue in the technology sector. Potential threats such as buffer overflow and injection have been commonplace in many commercial applications, leading to security and privacy concerns (Kuhn et al., 2017). For example, a buffer overflow attack can allow an attacker to corrupt the return address of a function and create undesired control flow. To counteract this, there are existing works that aim to detect when a program exhibits erroneous flow (Zeng et al., 2011; Yang et al., 2015). Before hitting a point where our program is forced to deal with its own vulnerabilities, we hope to detect and fix them. While hiring a programmer is certainly one solution, having analytic tools to locate and even fix vulnerabilities is a natural next step. To this end, several existing tools have been developed over the years, e.g., Vulpecker and Cppcheck (Li et al., 2016; Lipp et al., 2022). While these methods all aim to automate the task of vulnerability detection and repair, they are gated by their breadth, e.g., not task and language agnostic. There also exist deep learning and language model driven approaches, details of which can be found in the Related Work section. These generally require specifically curated and trained models to fix vulnerabilities.

In this paper, we aim to synthesise the power of large language models with the precedent for better tooling in the code vulnerability repair space. Our hypothesis is that down the line these approaches will be better as more scale allows models to learn and understand code better. We seek to only employ commercially available out-of-the-box LLM's like ChatGPT (OpenAI, 2023), Bard (Google, 2023), and Claude (Anthropic, 2023) for our experiments. With specific prompt engineering, we hope to increase performance without resorting to resource-intensive methods such as finetuning and limit our model access to only APIs. To our knowledge, finetuning existing LLMs is the current state of the art, demonstrated in (He & Vechev, 2023), but this approach can quickly become very costly. Although incomparable to state of the art benchmarks, prompt engineering has the potential to boost the performance of our models.

Another problem we tackle is the development of a semi-automatic evaluation tool that can evaluate whether a program is actually repaired. Many datasets pull publicly available commits and fixes from a wide variety of open source projects (He & Vechev, 2023; Bui et al., 2022). Although

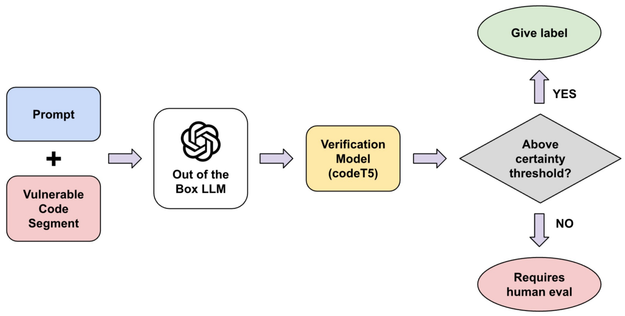

Figure 1. General Diagram for our methodology. We prepend a prompt to a vulnerable code segment and have the LLM rewrite the code with no vulnerabilities. This is passed to a classifier which gives it a label and an associated confidence score. We accept the label if it is above a certain threshold and human evaluate the rest of the samples.

these datasets are good at representing a variety of real-world code sources, it is difficult to create test cases as the code can only call locally defined functions and would require spinning up test cases for each project. This is clearly not feasible. Furthermore, many datasets have a proposed fix but there can exist multiple solutions to one problem. To this end, we propose a pairwise pretraining procedure for finetuning LLMs on zero shot vulnerability detection. This accompanied with manual human rating integrates more robustness into the rating and allows people to focus more attention on less samples.

We summarize our contributions as follows:

1. Evaluate and demonstrate the potential of various state of the art prompt engineering methods for vulnerability repair.
2. Created a zero shot vulnerability evaluation framework that leverages automatic evaluation with human-rating for robustness.

2. Threat Model

Our project takes several code snippets from a CWE dataset derived from various open-source projects and repositories. Each sample contains a specific CWE vulnerability. How this vulnerability can appear is through mistakes by software engineers and the writers of various commercial application. If these mistakes are gone undetected, an attacker could leverage them to disrupt the system. An attacker's exploit may lead to unintended behavior and potentially allow attackers to compromise the system, steal data, gain unauthorized access, or cause a denial of service. The goal of such an attack is provided by its context, or the type of CWE vulnerability. For example, a buffer overflow attack aims to manipulate the control flow and execute malicious code and a SQL injection could be used to delete or leak database entries. Although the vulnerabilities in many of the samples are not inherently designed for malicious purposes, compromising the code's integrity could potentially open the door to attacks.

3. Methods

3.1. Vulnerability Repair

For vulnerability repair, we prompt out of the box LLMs as shown in Figure 1. A typical evaluation would involve 1) taking a code segment, function, etc. 2) attaching meta data and other important features to our prompt 3) feeding it into the model 4) extracting and parsing the model output. In the case that our text exceeds the context length of the LLM, it is cut off. Furthermore, if the output is too long it is unfortunately cut off as well. This is an inherent limitation for all LLM-related research, not just this one.

Our method is contingent on an available API and entirely model agnostic. This makes it easy to extend to other LLMs besides OpenAI's model, e.g., Bard, Claude, etc. The repairs are also very fast, or as fast as inference is from these models. By not exposing model weights and limiting our method to an API, we are also protected against a series of white-box attacks.

3.2. Semi-Automatic Evaluation Model

In order to determine whether a vulnerability repair actually worked, we can reduce the problem to detecting whether a code segment is vulnerable or not. However, accomplishing this was not easy. Our first attempt was to rely on GPT-4 as our rater. However, when tested on a validation set, GPT-4 was accurate about 50% of the time, which is basically equivalent to random guessing.

Next, we depended solely on human rating, but this was absurdly slow. In order to properly rate examples, one must be very knowledgeable about the language and errors and in many cases there is a large degree of uncertainty. As shown in our preliminary evaluations in the Appendix, we had to label a lot of the samples as partially correct because of the ambiguity.

Finally, we decided to create an semi-automatic rater. This was done by taking the pretrained codeT5 model from Salesfore publicly available on Hugging Face (Wang et al., 2021). This model is trained on a wide variety of languages and code generation tasks, so we expect it to be knowledgeable of code semantics. All of our experiments used a small version of the original model due too out lack of computational resources. We removed the decoder portion of the model, replacing it with a MLP for binary classification. The training and model details are in the Appendix. We trained our model with the following loss function:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

$y_i \in \{0, 1\}$ denotes the ground truth label of the ith sample and $\hat{y} \in [0, 1]$ denotes the corresponding probability that the sample is vulnerable. Essentially, we are using traditional BCE Loss. Furthermore, we use AdamW with a learning rate of $5e-5$ for 100 epochs. Our modified model architecture (classifiction head) derives from the RoBERTa implementation on HuggingFace (Liu et al., 2019). Further details can be checked in our code.

As shown in Table 1 we found that our model with basic rounding had decent test accuracy but would not make us confident in our results. Therefore we enforced a threshold value to ensure the model only classifies high-confidence samples. That is, we can imagine if the model prediction for the ith samples is $\hat{y}_i$, then we only predict the sample given a threshold τ if:

$$\hat{y}_i > \tau \text{ or } 1 - \hat{y}_i > \tau$$

That is, if we are confident the sample is a positive or negative prediction. Otherwise we default to human rating. Although we did not explicitly verify, we believe that given more time, resources, and model feedback, a human could verify a few samples with high certainty (better than our classifier).

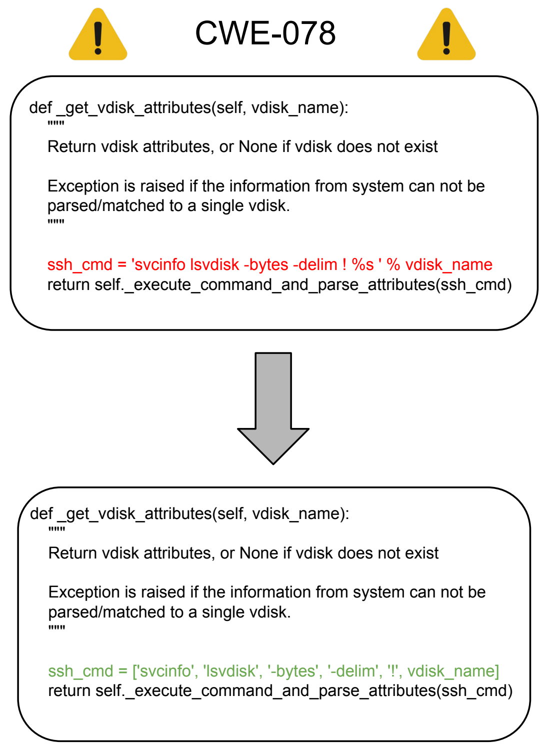

Figure 2. Example of CWE-078 vulnerability fixed by our Few Shot prompt. Top: Code with vulnerability and the line of interest in red. Bottom: Code with the vulnerability fixed (in green).

4. Experiment Details & Results

4.1. Datasets

We focused on 3 large datasets of CVEs: SVEN, Big-Vul, and Vul4J. Note that we classify vulnerabilities by the Common Weakness Enumeration (CWE), a standard list of software weakness types (He & Vechev, 2023; Bhandari et al., 2021; Fan et al., 2020; Bui et al., 2022).

SVEN. He & Vechev (2023) created a dataset with 9 different CWEs that are listed in MITRE top-25. More specifically, the ones we focus on include CWE-022 (path traversal), CWE-078 (improper neutralization), CWE-079 (cross-site scripting), CWE-089 (SQL injection), CWE-125 (out-of-bounds read), CWE-190 (integer overflow), CWE-416 (use after free), CWE-476 (NULL pointer dereference), and CWE-787 (out-of-bounds write). In every test case, we are provided the full definition of a function a given vulnerability and are tasked with fixing the function.

Big-Vul. Fan et al. (2020) involves C/C++ code vulnerabilities taken from the CVE dataset. Big-Vul contains contains 91 different vulnerability types, and in total 3,754 code vulnerabilities extracted from 348 open-source Github projects. The data set provides links to the repositories that had the vulnerabilities and also the commit IDs for before and after the vulnerability was fixed. Out of the 348 products, the top 3, Google Chrome, Linux, and Google Android account for 63.65% of the total collected commits and the top 10 account for 76.69% of the total collected commits. It is also interesting to note that the top three CWE types in the data set account for 33.94% of total commits and involve vulnerabilities related to data management. Those CWE types are CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-20 (Improper Input Validation) and CWE-125 (Out-of-bounds Read).

Vul4J. Vul4J is a dataset of real-world Java vulnerabilities (Bui et al., 2022). Each vulnerability in the dataset is provided along with a human patch, Proof-of-Vulnerability (PoV) test case(s), and other information for the reproduction of the vulnerability. Unfortunately, due to a coding issue, the we were unable to reproduce any of the vulnerabilities and had to trace each of the patches by hand.

4.2. Pure Human Eval Experiments (Preliminary)

We list our results in Table 3, 4, 5 and the corresponding prompts in Appendix B.1. In this setting, we used Zero Shot and Few Shot prompting; however, did not disclose the specific vulnerability we wanted to fix.

On the SVEN dataset, our models seems to do well on SQL injection and overall better with the basic prompt. Chain of Thought ultimately does not seem to help much. This is probably because SQL injection is more formulaic in nature and easily interpretable.

When prompted with samples from the Vul4J dataset, the model was able to successfully repair a simple function by replacing an if statement with a while loop. For more complex vulnerabilities, such as a robust XSS-attack, the model did not accomplish anything meaningful or significantly impactful in the repaired code. The outputted code was ultimately synonymous to the original code with some minor changes to its logic. We observed a notable difference in the model when switched to Chain-of-thought prompting: it was partially accurate in repairing 4 types of vulnerabilities. We define ”partially accurate” as the model properly classifying the type of vulnerability present in the code, while also displaying some level of intuition or effort to repair it–even if the solution itself is incorrect. We conclude that GPT-3.5 has the ability to perform well when following instructions or changing things that it is specifically told to do so. But it is unable to accomplish this on its own, even if the solution is readily available on the Internet. This observation motivated more formalized prompting methods for the model to lean on and increase its performance.

In the Big-Vul dataset, our results conclude that a handful of our samples were 100% accurate, but an even larger portion was still partially or entirely incorrect. The cases where the model was correct, the vulnerability could be fixed by modifying a single line of code within the function or making small changes like adding a simple bounds check. This may explain why GPT-3.5 was 100% accurate in these samples. In the partially incorrect case, the specific vulnerabilities were identified but the repair was unfortunately insufficient.

We were able to extract some meaningful information about the behavior of the GPT model as well as what sort of vulnerabilities it may be best equipped to handle. However, human rating is still extremely inconsistent and we were unsure about a lot of the outputs, motivating the development of a new set of experiments for our final report.

4.3. Semi-Automatic Rating Experiments

We present our results in Table 2 with prompts in Appendix B.2 on the SVEN dataset. Compared to the pure human eval experiment, we relax our assumptions by disclosing to the model what type of vulnerability to fix. (Note that an extension work could be to train a model to do this multiclass classification and then pass it to ChatGPT.)

What we find is that Few Shot COT was the best performing prompts, having around 44/82 (53.66%) repair rate. Overall, it seems like there was a lot of success with CWE-078, CWE-079, CWE-089, and CWE-125 while little to no success with CWE-022, CWE-416, CWE-476, and CWE-787. At least for OS command injection and SQL injection, the solutions were typically formulaic as shown in Figure 2. That is, it simply involved formulating the string and following known patterns to remove the vulnerabilities. In this specific case, we can see that wrapping the string as a list is the general solution and it recognizes it well. GPT models have excellent pattern recognition capabilities and our results highlight this. CWE-416 and CWE-476 with null pointer dereference are a bit more subjective. This is because there are many instances where broader context needed to figure out if a pointer can potentially be NULL as well as what fields in an object are even pointers. Because of a lack of context about the whole codebase, we found that at times the evaluation was unfair. There were extraneous details, functions, and other artifacts left out that could be critical to GPT's repair ability. Yet, we are only given a small function.

The biggest trade off between Zero and Few shot is the amount of context window that we are taking up when prompting the model. However, on the plus side for Few Shot learning, we can enrich our model with relevant examples that could improve its repair capabilities. We found in our experiments that Few Shot generally outperformed Zero Shot. In the Few Shot CoT experiment, we not only provided examples and told the model to think but crafted our own explanations for why we fixed the example in a certain way. Clearly, this helped the model reason about the task, recognizing what to focus on and why.

Compared to the pure human rating, we find a lot of similarities. Echoed in both results is the idea that CWE-089 and CWE-079 are well performing vulnerability fixes for LLMs.

5. Additional Studies

5.1. Line Level Repair

We implemented line-level security vulnerability prediction and repair on the VuL4J dataset. Our work is inspired by Fu & Tantithamthavorn (2022). We first prompt GPT-3.5 to rank the Top-10 vulnerable lines of code (1 as most vulnerable, 10 as least vulnerable) in a segment of code. We tested on samples of more than 15 lines in total length and if the model concluded that there were less than 10 vulnerable lines of code, instruct it to return a shorter list. This provided an interesting method for measuring the ”false-

Threshold	Fraction of Kept Samples ↑	Test Accuracy ↑	Test AUC ↑
0.5	**164/164**	0.7195	0.7195
0.6	133/164	0.7744	0.7750
0.7	117/164	0.8205	0.8184
0.8	100/164	0.8600	0.8587
0.9	93/164	0.8925	0.8932
*0.95	88/164	**0.9432**	**0.9424**

Table 1. We present our results on the test accuracy given various thresholds. *indicates the threshold we used for our experiments.

Method	CWE-022	CWE-078	CWE-079	CWE-089	CWE-125	CWE-190	CWE-416	CWE-476	CWE-787	Repair Rate ↑
Zero Shot (M)	0/3	5/10	1/4	19/20	2/6	0/1	1/1	0/1	0/1	-
Zero Shot (H)	0/3	0/1	1/1	0/0	2/9	0/4	0/6	0/7	0/4	-
Zero Shot (Total)	0/6	5/11	2/5	19/20	4/15	0/5	1/7	0/8	0/5	31/82
Zero Shot COT (M)	0/3	3/10	1/4	17/20	1/3	1/0	0/1	0/1	0/0	-
Zero Shot COT (H)	0/3	1/1	1/1	0/20	5/12	0/5	0/6	0/7	1/5	-
Zero Shot COT (Total)	0/6	4/11	2/5	17/20	6/15	1/5	0/7	0/8	1/5	31/82
Few Shot (M)	0/4	8/10	1/4	19/19	0/4	1/2	0/1	0/1	0/1	-
Few Shot (H)	0/2	1/1	1/1	1/1	3/11	1/3	2/6	1/7	1/4	-
Few Shot (Total)	0/6	9/11	2/5	20/20	3/15	2/5	2/7	1/8	1/5	40/82
Few Shot COT (M)	0/3	9/11	1/3	20/20	2/4	1/2	0/1	0/1	1/1	-
Few Shot COT (H)	1/3	0/0	2/2	0/0	3/11	1/3	1/6	1/7	1/4	-
Few Shot COT (Total)	1/6	9/11	3/5	20/20	5/15	2/5	1/7	1/8	2/5	**44/82**

Table 2. We present our results for the dataset given in the SVEN paper. We test four kinds of prompts with details in Appendix B.2 on GPT-3.5-turbo. The results indicate the fraction of correctly generated vulnerability repairs.

positive rate" of the model, namely, how many lines of code it misclassifies and ranks as a vulnerable line of code before correctly detecting the first line of code containing the actual vulnerability. We defined correctness as at least 1 vulnerable line of code appearing in its ranking, regardless of the placement. Next, we asked the model to repair each line of vulnerable code in the list. For this side project, we focused solely on CWE-089 and CWE-022. The types of vulnerabilities chosen were not arbitrary: we leverage the findings in our original experiment and picked vulnerabilities the LLM performed significantly well/worse on. CWE-089 repair typically requires one line fixes, a suitable contender for line-level repair studies, while CWE-022 requires a more complex approach, heavily dependent on the context of the entire program–this foreshadows the model performing poorly for this type of vulnerability.

Our previous discussion seems to reflect in our results. We worked with a small sample size of 10 samples for each type of vulnerability. 14/20 were correctly predicted with at least 1 line of vulnerable code in the ranking: 8/10 CWE-089, 6/10 CWE-022. The false-positive rate was < 1 incorrectly labeled lines for CWE-089, and 3-4 lines for CWE-022. Out of the 8 CWE-089 samples, 7 were correctly repaired (in particular, only in cases where the vulnerable line was ranked first). The model was entirely unsuccessful for implementing a line-level repair for CWE-022.

The results from this small experiment revealed competitive performance to our original experimental design for fixing CWE-089 vulnerabilities, with equally poor results for CWE-022. (A future study may reduce the margin of error by only feeding the model isolated lines of vulnerable code). But there are some flaws to this approach that make its real-world application still pre-meditated. The possibility of false-positives in line-level prediction and repair requires human evaluation to correctly identify which lines contain the vulnerability and therefore accept the suggested repair, which is counter-intuitive. It is also difficult to make assumptions on the model's understanding of the overall program since it conducts fine-grained analysis line-by-line.

5.2. False Positive

We devised a false positive study comprising two distinct prompts, presented in Appendix C, to assess GPT's behavior and accuracy when presented with fixed code snippets. We again used the SVEN dataset consisting of functions both before and after a CWE vulnerability was addressed through human intervention. We provided GPT 3.5 with the first prompt accompanied by the fixed version of the code. In almost all cases GPT 3.5 still detected additional potential vulnerabilities and attempted to fix them. In the majority of instances those fixes included minor changes to the code that were not necessary to fix the CWE that was present in the function before the fixed version. But those changes did not add incorrect behavior and made the code more robust by adding protection to other potential vulnerabilities, so we did not consider those instances as false positives. However, we observed false positives in 3 out of 11 instances. These false positives entailed unnecessary or incorrect changes to the code, sometimes also altering its functionality. In some instances where GPT hallucinates, it only altered the style of the code or rewrote the code in an alternative way and stated that it fixed potential vulnerabilities. An example of this behavior can be found in Appendix C, where GPT rewrote the code and claimed to have fixed vulnerabilities but it only changed the use of a data structure by using a standard library data structure instead of using the custom data structure Array2D from the original code. For the second prompt, once again the fixed version of the code was provided. Under this prompt, GPT demonstrated a tendency to rewrite code, regardless of the vulnerability's presence. Out of 20 instances, we observed 8 false positives, indicating instances where GPT incorrectly identified a vulnerability that had been previously addressed and rewrote code attempting a fix. The advantage of this prompt was that GPT focused on the CWE in question, which avoided many unnecessary changes that it would perform when it was given the first prompt. Across both prompts, GPT showcased a propensity to make changes to the code, with alterations occurring in 28 out of 31 instances. This behavior suggests an inclination towards generating new code rather than solely addressing the specified vulnerability. There were only 3 out 31 instances where GPT stated that the vulnerability had been addressed and did not rewrite code. When we conducted this study we prompted GPT multiple times with the same code snippets in random order in order to check for different and inconsistent behavior. When doing this we made an interesting observation regarding a specific and inconsistent behavior that occurred in some instances when we used the second prompt. When GPT claimed that the vulnerability had been addressed for a code snippet it would then conclude for almost all other code snippets that the vulnerability had been addressed as well. This behavior would always occur after GPT was prompted with prompt 2 and concluded that the vulnerability had been addressed. To further test this we prompted GPT with the version of the function that contained the vulnerability, and it would still lead to this undesirable behavior. This behavior is problematic since it can lead to false negatives as well. Our ablation study highlights the nuances and challenges associated with leveraging GPT for security vulnerability detection and remediation. While GPT exhibits promising capabilities, evidenced by its attempts to rectify vulnerabilities, it also demonstrates a tendency towards false positives and code alterations that may not align with the intended fix. These findings underscore the importance of further research and refinement to enhance GPT's effectiveness in security related tasks. Further research needs to be conducted regarding the special case that would lead to false negatives.

5.3. Adversarial Attacks

We performed adversarial attacks based on the SVEN datasets, notably vulnerability CWE 190 and CWE 416. These attacks were formulated through the following methods: changing the prompt that was fed in, inserting comments that were irrelevant to the vulnerabilities, and inserting irrelevant code. For the first method, the prompt input method would be: ”Find * vulnerability in the below code and explain how to fix it...” where * is a vulnerability that does not exist in the code. We find that this method of attack performs very well as the model manages to ”hallucinate” and trust in the prompt's judgement rather than to make a different assumption. This may stem from the fact that it does not have external knowledge of the overall functionality of the code, so it assumes that the prompt is correct in this regard. For the second method, comments were added throughout the code either to mark out vulnerabilities or attempt to invalidate a portion of the code, even if the code itself was unchanged. This method was rather ineffective, as Chat-GPT completely disregarded all of the comments. For the last method, the prompt would incorrectly describe a type of vulnerability and ask the model to look for an example of it in the given code. This performs similar to the first method mentioned previously.

We also prompted a combination of the three methods, but any resulting method that had either 1 or 3 would always output something random that failed to repair the intended vulnerability.

6. Challenges

We foreseen some limitations to our project early on in the quarter. Some of these are discussed below with our proposed solution in the final experimental design:

- One threat to this project is that the training data for ChatGPT may contain the bugs or fixes in the datasets we will be using. This threat exists for all LLM-related research, not just this one. But this threat is less of a concern since LLMs do not see the pair of bugs and their fixed code during training, and their training data often contains at most the buggy code or the fixed code, but not both. We concluded that this limitation had a minor, untraceable effect to our project (to our knowledge).
- Examining 3 large datasets may be exhaustive. We do not intend to test on the entirety of each dataset. Our emphasis on testing on multiple datasets is for validating the robustness and generalizability of our method. By evaluating its performance across various datasets, we can ascertain its effectiveness under different conditions and ensure that any observed patterns or improvements are not merely artifacts of a specific dataset. Our original experiment heavily relied on the SVEN dataset and the remaining ones were used to supplement the preliminary results and partially apply to one of our ablation studies.

- Human rating was a major constraint to the pace of our project. It also introduced an added layer of error depending on a person's understanding of programming, which may hinder their ability to properly assess the repair. We proposed a semi-automatic rating experimental design to counteract this, which still relied partially on human graders, but ultimately accelerated the entire process by integrating a pretrained model for automated repair evaluation.
- A lack of context was also an issue. Some functions, classes, and variables were not provided at the granularity we were looking at making it almost unfair to try and repair it.

7. Related Work

Recent program repair methods are expanding on large language models. It was discovered that the performance of LLMs is competitive to common deep learning approaches and notably better than the results reported for standard program repair approaches (Jiang et al., 2023). In particular, finetuning the LLM with general APR data greatly improves its vulnerability-fixing capabilities. However, it is important to note that there are several limitations on LLM-based APR–some of which are completely indicative of the existing challenges posed by current APR techniques. Large language models and APR techniques, except Codex, only fix vulnerabilities that require simple changes, such as deleting statements or replacing variable/method names (Wu et al., 2023). However, the performance of LLMs can be optimized through its unique framework.

In contrast to previous methods, LLMs offer a dialogue system in which further information, e.g., the expected output for a certain input or some context for a segment of code, can be entered. By providing such hints to the LLM, its success rate can be further increased, outperforming state-of-the-art (Sobania et al., 2023). This introduces us to field of prompt engineering. Zero-shot and chain-of-thought learning are the two most basic approaches for prompting the model, pioneered by many LLM papers and commonly used for benchmarking LLM performance (Weng, 2023). Zero-shot or basic prompting simply feeds the model the task and asks for the results. Chain-of-thought prompting expands on zero-shot by asking the model to describe the step-by-step process that eventually leads to the final answer (Wei et al., 2022). There are a few existing works that explore harnessing LLMS for vulnerability repair (Pearce et al., 2023; Islam et al., 2024) that use reward functions and other methods to finetune models for the task. However, to our knowledge, none have pushed the limits of prompt tuning for this task.

We would also like to mention non-neural-network approaches for automated program repair. Static code analysis is often used to scan source code for security vulnerabilities. While static C analyzers perform well in benchmarks with synthetic bugs, it misses more than half of vulnerabilities in a benchmark set of real-world programs (Lipp et al., 2022). Therefore static code analysis is one of the less novel methods for program repair. Many bugs hence remain undetected, especially those beyond the classical memory-related security issues.

8. Future Work

- One area of future work is to develop better performing methods for automatically telling if a piece of code is vulnerable or not. An easy way is to scale the model we currently have to the larger size. Furthermore, we could have done more hyperparameter search given more time and resources.
- There are a multitude of different prompts that we can use for generating repairs. Lately, there has been the development of Tree of Thought prompts (Yao et al., 2024), which could further propel our performance.
- Expanding upon our additional studies. For line-level automated security vulnerability repair, vulnerability prediction may not be very opportunistic, but line-level repair has shown remarkable promise for types of CWE with a line-level scope by nature.

9. Conclusion

To reiterate, our main experiment that implemented semi-automatic rating highlighted that Few Shot COT was the best performing prompts with a repair rate of over 50%. We tested on 9 types of vulnerabilities: CWE-022, CWE-078, CWE-079, CWE-125, CWE-190, CWE-416, CWE-476, and CWE-787. Furthermore, we presented a series of ablation studies to evaluate the robustness and facilitate more understanding about our results.

The current landscape of security vulnerability repair tends to be more reactive than proactive, with processes often mobilized in response to the discovery of threats rather than preempting them. This reactionary stance, while practical in the short term, may not suffice in the face of rapidly evolving cyber threats and the increasing complexity of modern software systems. Our research leverages out of the box LLMs to automate the process of identifying and rectifying security vulnerabilities early in the development process. By integrating automated vulnerability detection and repair processes into the initial stages of software development, we aim not only to enhance the security of software applications but also to significantly reduce the time and resources currently expended on post-deployment fixes. We would also like to highlight that our proposed methods makes use of very minimal resources and is very feasible with just an API and some computing credits.

References

Anthropic, 2023. URL https://www.anthropic.com/news/introducing-claude.

Bhandari, G., Naseer, A., and Moonen, L. CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21)*, pp. 10. ACM, 2021. ISBN 978-1-4503-8680-7. doi: 10.1145/3475960.3475985.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020.

Bui, Q.-C., Scandariato, R., and Ferreyra, N. E. D. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pp. 464–468, 2022. doi: 10.1145/3524842.3528482.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Fan, J., Li, Y., Wang, S., and Nguyen, T. N. A c/c++ code vulnerability dataset with code changes and cve summaries. MSR '20, pp. 508–512, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177. doi: 10.1145/3379597.3387501. URL https://doi.org/10.1145/3379597.3387501.

Fu, M. and Tantithamthavorn, C. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pp. 608–620, 2022. doi: 10.1145/3524842.3528452.

Google, 2023.

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

Islam, N. T., Khoury, J., Seong, A., Parra, G. D. L. T., Bou-Harb, E., and Najafirad, P. Llm-powered code vulnerability repair with reinforcement learning and semantic reward. *arXiv preprint arXiv:2401.03374*, 2024.

Jiang, N., Liu, K., Lutellier, T., and Tan, L. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1430–1442, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi: 10.1109/ICSE48619.2023.00125. URL https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00125.

Kuhn, D., Raunak, M., and Kacker, R. It doesn't have to be like this: Cybersecurity vulnerability trends. 2017-12-22 2017. doi: https://doi.org/10.1109/MITP.2017.4241462.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., and Hu, J. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pp. 201–213, 2016.

Lipp, S., Banescu, S., and Pretschner, A. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pp. 544–555, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534380. URL https://doi.org/10.1145/3533767.3534380.

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

OpenAI, 2023. URL https://openai.com/chatgpt.

Pearce, H., Tan, B., Ahmad, B., Karri, R., and Dolan-Gavitt, B. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2339–2356. IEEE, 2023.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

Sobania, D., Briesch, M., Hanna, C., and Petke, J. An analysis of the automatic bug fixing performance of chatgpt, 2023.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Weng, L. Prompt engineering. *lilianweng.github.io*, Mar 2023. URL https://lilianweng.github.io/posts/2023-03-15-prompt-engineering/.

Wu, Y., Jiang, N., Pham, H. V., Lutellier, T., Davis, J., Tan, L., Babkin, P., and Shah, S. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '23. ACM, July 2023. doi: 10.1145/3597926.3598135. URL http://dx.doi.org/10.1145/3597926.3598135.

Yang, S., Yan, D., Wu, H., Wang, Y., and Rountev, A. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pp. 89–99. IEEE, 2015.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Zeng, B., Tan, G., and Morrisett, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pp. 29–40, 2011.

A. Additional Results

Method	Model	CWE-022	CWE-078	CWE-079	CWE-089
Basic Prompt	GPT-3.5-turbo	1/6	5/11	0/5	18/20
Chain-of-Thought	GPT-3.5-turbo	1/6	1/11	0/5	14/20

Table 3. We present our results for the dataset given in the SVEN paper. We test two kinds of prompts with details in Appendix B on GPT-3.5-turbo. The results indicate the fraction of correctly generated vulnerability repairs.

Method	Model	100% accurate	Partially accurate	incorrect
Basic Prompt	GPT-3.5-turbo	1/20	4/20	15/20
Chain-of-Thought	GPT-3.5-turbo	1/20	8/20	11/20

Table 4. We present our results for the dataset given in the Vul4J paper. We test two kinds of prompts with details in Appendix B on GPT-3.5-turbo.

Method	Model	100% accurate	Partially accurate	incorrect
Basic Prompt	GPT-3.5-turbo	4/20	7/20	9/20

Table 5. We present our results for the dataset given in the Big-Vul paper.

B. Prompts

B.1. Human Eval

```
Please rewrite the following code and fix the security vulnerability.
Only include code in your output.
Function Name: *func_name*
Code: *func_src_before*
```

Figure 3. General version of the Basic Prompt used for human evaluation experiments for our midterm report.

```
Let's think about this problem step by step.
You are given a function name and its code as shown below.
First find the possible vulnerabilities and then rewrite the code to fix
the vulnerabilities.
Only output the code.
Function Name: *func_name*
Code: *func_src_before*
```

Figure 4. General version of the Chain-of-Thought Prompt used for human evaluation experiments for our midterm report.

B.2. Semi-Automatic Eval

```
Please rewrite the following code and fix the security vulnerability.
Only include code in your output.
Function Name: *func_name*
Vulnerability: *vul_type*
Code:
*func_src_before*
```

Figure 5. Zero Shot used for Semi-Automatic Eval.

```
Let's think about this problem step by step. You are given a function name and
its code as shown below.
First find the possible vulnerabilities and then rewrite the code to fix
the vulnerabilities.
Only output the code.
Function Name: *func_name*
Vulnerability: *vul_type*
Code:
*func_src_before*
```

Figure 6. Zero Shot COT used for Semi-Automatic Eval.

```
Please rewrite the following code and fix the security vulnerability.
Here are a few examples.
Function Name: create_basename_core
Vulnerability: cwe-022
Code:
def create_basename_core(basename):
    try:
        basename = basename.casefold()
    except Exception:
        basename = basename.lower()

    basename = basename.replace(' ', '-')
    basename = re.sub(r'<[^>]*>', r'', basename)
    basename = re.sub(r'[^a-z0-9\\-]', r'', basename)
    basename = re.sub(r'\\-\\-', r'-', basename)
    basename = urllib.parse.quote_plus(basename)

    return basename

Answer:
def create_basename_core(basename):
    try:
        basename = basename.casefold()
    except Exception:
        basename = basename.lower()

    basename = re.sub(r'[ \./]', r'-', basename)
    basename = re.sub(r'<[^>]*>', r'', basename)
    basename = re.sub(r'[^a-z0-9\\-]', r'', basename)
    basename = re.sub(r'\\-\\-', r'-', basename)
    basename = urllib.parse.quote_plus(basename)

    return basename
Function Name: *func_name*
Vulnerability: cwe-022
Code:
*func_src_before*

Answer:
```

Figure 7. Few Shot used for Semi-Automatic Eval (cwe-022 vulnerability).

```
Please rewrite the following code and fix the security vulnerability.
Here are a few examples.
Let’s think step by step about this problem.
Function Name: create_basename_core
Vulnerability: cwe-022
Code:
def create_basename_core(basename):
    try:
        basename = basename.casefold()
    except Exception:
        basename = basename.lower()

    basename = basename.replace(' ', '-')
    basename = re.sub(r'<[^>]*>', r'', basename)
    basename = re.sub(r'[^a-z0-9\\-]', r'', basename)
    basename = re.sub(r'\\-\\-', r'-', basename)
    basename = urllib.parse.quote_plus(basename)

    return basename

Explanation: The issue in the code lies in the improper sanitization of basename.
That is, we do not check for inputs that match the pattern './', which could cause
path traversal vulnerabilities. Hence, we should add a regex statement for replacing
any occurence of this pattern in basename.
Answer:
def create_basename_core(basename):
    try:
        basename = basename.casefold()
    except Exception:
        basename = basename.lower()

    basename = re.sub(r'[ \./]', r'-', basename)
    basename = re.sub(r'<[^>]*>', r'', basename)
    basename = re.sub(r'[^a-z0-9\\-]', r'', basename)
    basename = re.sub(r'\\-\\-', r'-', basename)
    basename = urllib.parse.quote_plus(basename)

    return basename
Function Name: *func_name*
Vulnerability: cwe-022
Code:
*func_src_before*

Explanation:
Answer:
```

Figure 8. Few Shot used COT for Semi-Automatic Eval (cwe-022 vulnerability).

C. False Positive Study Material

```
//Fixed function from SVEN dataset that previously contained CWE 476
generatePreview (const char inFileName[],
		float exposure,
		int previewWidth,
		int &previewHeight,
		Array2D <PreviewRgba> &previewPixels)
{
    //
    // Read the input file
    //

    RgbaInputFile in (inFileName);

    Box2i dw = in.dataWindow();
    float a = in.pixelAspectRatio();
    int w = dw.max.x - dw.min.x + 1;
    int h = dw.max.y - dw.min.y + 1;

    Array2D <Rgba> pixels (h, w);
    in.setFrameBuffer (ComputeBasePointer (&pixels[0][0], dw), 1, w);
    in.readPixels (dw.min.y, dw.max.y);

    //
    // Make a preview image
    //

    previewHeight = max (int (h / (w * a) * previewWidth + .5f), 1);
    previewPixels.resizeErase (previewHeight, previewWidth);

    float fx = (previewWidth  > 1)? (float (w - 1) / (previewWidth  - 1)): 1;
    float fy = (previewHeight > 1)? (float (h - 1) / (previewHeight - 1)): 1;
    float m  = Math<float>::pow (2.f, IMATH_NAMESPACE::clamp (exposure + 2.47393f, -20.f, 20.f));

    for (int y = 0; y < previewHeight; ++y)
    {
    for (int x = 0; x < previewWidth; ++x)
    {
        PreviewRgba &preview = previewPixels[y][x];
        const Rgba &pixel = pixels[int (y * fy + .5f)][int (x * fx + .5f)];

        preview.r = gamma (pixel.r, m);
        preview.g = gamma (pixel.g, m);
        preview.b = gamma (pixel.b, m);
        preview.a = int (IMATH_NAMESPACE::clamp (pixel.a * 255.f, 0.f, 255.f) + .5f);
    }
    }
}
```

Figure 9. Example of ground truth SVEN CWE-476 that is already fixed.

```cpp
//GPT "Fix" changes functionality by using std vector instead of Array2D custom type

#include <vector>
#include <algorithm> // for std::max

void generatePreview(const char inFileName[],
                     float exposure,
                     int previewWidth,
                     int &previewHeight,
                     std::vector<std::vector<PreviewRgba>> &previewPixels) {
    // Read the input file
    RgbaInputFile in(inFileName);

    Box2i dw = in.dataWindow();
    float a = in.pixelAspectRatio();
    int w = dw.max.x - dw.min.x + 1;
    int h = dw.max.y - dw.min.y + 1;

    std::vector<std::vector<Rgba>> pixels(h, std::vector<Rgba>(w));
    in.setFrameBuffer(ComputeBasePointer(&pixels[0][0], dw), 1, w);
    in.readPixels(dw.min.y, dw.max.y);

    // Make a preview image
    previewHeight = std::max(static_cast<int>(h / (w * a) * previewWidth + 0.5f), 1);
    previewPixels.resize(previewHeight, std::vector<PreviewRgba>(previewWidth));

    float fx = (previewWidth > 1) ? static_cast<float>(w - 1) / (previewWidth - 1) : 1;
    float fy = (previewHeight > 1) ? static_cast<float>(h - 1) / (previewHeight - 1) : 1;
    float m = Math<float>::pow(2.f, IMATH_NAMESPACE::clamp(exposure + 2.47393f, -20.f, 20.f));

    for (int y = 0; y < previewHeight; ++y) {
        for (int x = 0; x < previewWidth; ++x) {
            PreviewRgba &preview = previewPixels[y][x];
            const Rgba &pixel = pixels[static_cast<int>(y * fy + 0.5f)][static_cast<int>(x * fx + 0.5f)];

            preview.r = gamma(pixel.r, m);
            preview.g = gamma(pixel.g, m);
            preview.b = gamma(pixel.b, m);
            preview.a = static_cast<int>(IMATH_NAMESPACE::clamp(pixel.a * 255.f, 0.f, 255.f) + 0.5f);
        }
    }
}
```

Figure 10. Example of False positive that is passed through ChatGPT.

```
Please rewrite the following code and fix the security vulnerability.
code:
```

Figure 11. Prompt 1 used for false positive study.

```
The following code had a CWE cwe_number vulnerability and a human fix was performed.
Does the following code still have the vulnerability?
Only rewrite code if vulnerability is still present.
Code:
```

Figure 12. Prompt 2 used for false positive study.