Examining Zero-Shot Vulnerability Repair with Large Language Models

The paper deals with an LLM model's ability to repair Large Language Models. The key factor that is of interest is that LLM would be in a zero-shot setting that is the LLM would be tasked with something that it has not encountered before or trained for. Then, there are a couple of questions of interest that the authors try to answer through their research and this paper. Can the model generate safe and functional code to replace the vulnerable code? The emphasis would be to ensure that the repaired code preserves the intended original functionality. Another question is whether changing the information provided through the prompts affects the model's ability to suggest fixes. It should also be researched on the challenges this strategy of fixing vulnerabilities would face in the real world, and if this approach is reliable.

The researchers experiment with 6 LLMs of which 5 are commercially available and 1 is open-source. All of the LLMs are tested on zero-shot settings on synthetic, hand-crafted, and real-world vulnerabilities with different prompts and LLM settings. The suggested code by LLM is tested for functionality and evaluated for security. The authors also did a model parameter sweep to identify the good parameters for the model such as the temperature and top_p.

It was found that different settings are useful for fixing different vulnerabilities. For example, higher temperature performed well at fixing buffer overflows while it didn't perform as well at SQL Injection vulnerabilities. Also, it was observed that different models performed at different efficiency which is to be expected as they would be trained with different datasets and different parameters. For example, cushman was found to work better on C code whereas davinci works better on Python code. Further, the authors found that the prompts affected the performance of the model in suggesting secure and functional code. It was mostly seen that the model gave better results when provided with code of bigger length providing more context and information. Another observation was that the models were not proficient in suggesting code for hardware vulnerabilities in the domain of Verilog. The major conclusion that the authors drew from the results is that "any one correct code completion is all that is required to fix a given bug, and all scenarios were successfully repaired by at least one combination of template and engine."

The question of whether the model would perform well in real-world scenarios was tested out by the authors by investigating real-world scenarios from public open-source projects. However, since the real-world code is larger the authors localized the bugs by reducing the code to the areas that specifically have vulnerabilities. Based on these experiments it was noted that an LLM was able to repair 8/12 projects which are not bad in comparison to the state-of-the-art non-LLM solution, ExtractFix which was able to patch 10 out of 12 projects. However, the patches are still implausible to an extent because it introduces new vulnerabilities to the code. Out of the 117 simple synthetic programs which had 58,500 patches, 3,688 patches were made by the model. On the handcrafted model, 2,796 patches were made out of 10,000 possible patches.

Discussion

- The paper focuses on the result that almost all vulnerabilities were patched by at least one model. The question then arises whether this claim is practical or impressive. Again, the missing part of the puzzle is the context in which this claim should be viewed. This paper is one of the first papers in this direction and should not be held to the same standards or level of expectations as papers in other well-established domains. The primary goal of the claim is to state that LLM has the power to accomplish the task, and hence with further research, it is possible that a single LLM might be able to patch more vulnerabilities making the approach more practical.
- Another point of difference between this paper and the ones presented at the beginning of the class is that while the former presentations were more well-established and structured with an accepted way of doing things, this paper is more subjective. This could especially be observed in the evaluation where the analysis of the results is very subjective, whereas in the first papers there were well-defined and accepted metrics of evaluation in each domain.
- Another approach that the authors took that is debatable is the localizing of vulnerabilities to test the LLMs against real-world vulnerabilities. Even though this approach still tests the model's ability to fix a bug, it takes away the evaluation of whether the model would be able to find the bug from a large piece of code. In other words, localizing the vulnerabilities and testing it ensures only that the model can perform well when the vulnerable code to total code ratio is high.
- A giant assumption that was tested through the efforts of the authors was whether an LLM model that does not really understand the code that it is outputting is able to fix vulnerable code. That is, if the model is able to create code does that automatically mean that it would be able to fix code as well? The model basically would just be using statistics in the same way it does when creating code to try and patch vulnerabilities, but this lack of understanding limits the model's power. For example, the model would not have the capability to fix logical bugs/vulnerabilities such as deadlocks. The future of LLMs would be to pair the model with logical reasoning models to fix this gap in their abilities.
- The hard reality of the software engineering world is that even though programmers are supposed to care about security and secure coding, they seldom do as their focus is primarily on functionality. This is where tools such as the one presented in the paper would come in handy to make suggestions pointing towards more secure ways of achieving what the programmer is trying to achieve. The discussion arose about whether it would standardize the way of coding, as the same model would suggest a similar style of coding to all users. However, more than the LLM dictating the way code would be written, it is the other way that happens. The LLM imitates how the majority of code is written and suggests it.
- The role of humans in a world where models such as LLM take up the bulkier parts of programming would be in areas like verifying the programs, training such models, or even in being good at using the correct prompts to get the required results out of such tools.
- The importance of tests such as regressive testing in these kinds of scenarios was also discussed as patching up a certain section of the code in a certain way, could introduce a lot of other

vulnerabilities in other areas. Moreover, if there is a vulnerability in the process, that is the code suggestion tool, then the vulnerability would get reintroduced every time.