Employing LLM-Based Evaluation for Memory Safety Bug Detection and Resolution C

CS 487 Secure Computer Systems Final Report Fall 2024 University of Illinois at Chicago Professor Xiaoguang Wang

Dawid Biel (djbiel2@uic.edu), Robert D. Hernandez (rherna70@uic.edu)

Context

Problem

Memory safety bugs are widely impactful for security of computer systems. Memory safety bugs can be particularly dangerous because they can be exploited by attackers to:

- Execute arbitrary code
- Steal sensitive data
- Bypass security mechanisms
- Gain elevated privileges
- Crash or compromise the entire system

Can we use LLMs to automatically suggest memory safety bug fixes using only static analysis, and can we begin to compare LLM output using similar or identical prompt and data input?

Goals

Begin the utilization of LLMs for static analysis of C program code for common types of memory safety bugs and develop a common format for direct comparison of LLM performance across bug types

Tools

- LLMs
  - ChatGPT4o
  - LLAMA3_1
- CVEs
  - 15 CVEs analyzed
- System and User Prompts
- REST API invocation using Python3 & devcontainers
- CoPilot for Cover Page image Generation

Experiment

For this project we chose to have multiple LLMs analyze code for memory errors. Through our initial discussions we decided to have a structured approach and to take an iterative approach to a feasibiltiy study to determine if our goal of LLM fix generation via static analysis (the exclusion of runtime context) was possible. Choosing three memory-safety related bug types we gathered a set of recent CVEs containing the bug types (also known as CWEs): Use-After-Free, Double-Free and Buffer Overflow.

To establish a baseline we agreed on having two LLMs analyze our data using the same prompt verisons: ChatGPT4o and LLAMA_3.1. While other models were considered for inclusion, this decision was made as they both have accessible and relatively capable models requiring minimal supporting setup.

We began by identifying potential memory safety bugs via the MITRE Database <cwe.mitre.org> and search for the three types. CWE stands for Common Weakness Enumeration, and it is a collection of all the different types of weaknesses in code. We collected these in the project repository under `src/`. Noting that buffer overflow memory safety bugs were the most prevalent with over sixteen thousand records. Clearing conversation history for each iteration, we tested and verfied each LLMs were capable of simply identifying a set of obvious examples of each type of memory safety bug.

Having confirmed our sample, and identified our bugs for examination, we prompted the models and refined the prompt to have the following (**the code block is intentially empty as it this document** and would typically contain the buggy code sample, found under `data/` in the project repository)

- LLM Input
    - Prompt v3 "You are a helpful AI assistant familiar with the C programming language, cybersecurity and low level memory safety bugs. Construct your answers using concise language, and do not add additional data or make up answers.

      "Evaluate the following code for memory-related errors, focusing on buffer overflows, double free occurrences, and use-after-free vulnerabilities. Identify any areas in the code where memory handling could lead to security vulnerabilities or instability in execution. Provide an analysis that includes specific lines or functions prone to these issues, and suggest corrections if needed to ensure safe memory allocation, access, and deallocation.

      ```
      ```

    - Code
        - We used a selection of 15 CVEs in the C programming language, mostly from 2024. Located at `src/RealLifeExamples/` in the project code repository.

After gathering all responses, compared them using metrics we defined such as accuracy, solution quality, comprehensibility etc. We picked one CVE in the Linux Kernel's libaom: Heap Buffer Overflow CVE-2024-5197 to perform an additional layer of sentiment analysis (project repo `docs/Results/buffer_overflow_CVE-2024-5197.md`) comparing the performance of the LLMs using ChatGPT4o in additon to our sentiment.

Ensuring that while gathering data to codenamed each LLM so as not to introduce bias when when reading the responses.

The LLMs provided responses starkly different in their analysis style. ChatGPT4o was very structured in its response approach in that it first identified the issue, then suggested fixes, showed affected code, showed revised code, gave improvements, as well as a summary. LLAMA on the other hand had everything in a paragraph writeup and sometimes it gave the full corrected code.

Overall, ChatGPT4o was much better at recognizing the vulnerabilities and implementing fixes. It also shined when it came to longer code. We specifically chose a longer piece of code and LLAMA was unable to process it, while ChatGPT4o recognized the bug. LLAMA only shined when it came to its references at the end which ChatGPT4o did not include. We did not specifically ask it to provide them, but it was interesting to see it give them to us.

Challenges

Sourcing CVEs:

"One of my main difficulties was finding the site mentioned above and getting references to sites where I was able to find the actual vulnerability. One of my main goals was to get a wide range of vulnerabilities. I wanted to get a few examples from each decade but that turned out to be difficult. For older vulnerabilities, majority of references point to websites that either no longer exist or ger redirected to the main page. For newer ones, I was restricted by the website from viewing any parts due to security complications of it still being exploited. I spend much time trying to get any sample code for (CVE-2024-0012) affecting Palo Alto

Networks PAN-OS Software, which came out as a 9.8 score and was widely talked about in security circles, but I was unsuccessful. I did, however, find much more luck in open source projects where the bugs were clearly shown and I had access to the entire program file. With that discovery we chose to pick majority of the memory bugs from 2024, to further the point that these are still extremely prevalent and that a LLM can detect and remediate them. Collecting five sample of each and stored them in our Github repository." - Dawid Biel

Data

```
We have 5 Buffer Overflows
    CVE-2024-5197
    CVE-2024-9915
    CVE-2024-42546
    CVE-2024-49777
    CVE-2024-49895
We have 5 Double Frees
    CVE-2024-50235
    CVE-2024-50152
    CVE-2024-50159
    CVE-2024-50215
    CVE-2024-50235

We have 5 Use-After Frees
    CVE-2022-20141
    CVE-2022-2621
    CVE-2021-0920
    CVE-2010-4168
    CVE-2010-2941
```

Conclusion

Based on initial findings, it seems feasible that currently available LLMs are suffeciently capable with only static input to reason about memory safety bugs and generate potential fixes. As our prompt was minimally refined to produce expected behavior from both LLMs this would imply non-insignifcant performance improvement through prompt engineering is possbile independent of LLM choice, and the addtion of runtime input (crash logs, fuzzer output) not neccesarily a pre-requisite to functional automatic fixes.

Future Work

- Automatic Fix Testing
- Additional LLM evaluation: Google BERT
- Integration of runtime input (crash logs, fuzzer output, etc.)
- Refinement of LLM prompts for machine readable output

Glossary

- LLM: Large Language Models https://en.wikipedia.org/wiki/Large_language_model
- CVE: Common Vulnerabilities and Exposures

- CWE: Common Weakness Enumeration
- ChatGPT4o - OpenAI LLM
- LLAMA_3.1 - Meta LLM
- BERT - Google Family of LLMs

References

- NIST https://nvd.nist.gov/
- Chromium Bug Tracker https://issues.chromium.org/issues
- MITRE CVE Database https://cve.mitre.org/
- Project private Github repository https://github.com/Rhernandez513/cs487-proj