Project Proposal for COMP517: Advanced Operating Systems

Abhinav Jain, aj70 [1] **Jie-Si Chen, jc182** [1]

1. Introduction

In today's digital age, the ubiquity of software applications in everyday life has magnified the importance of detecting and rectifying vulnerabilities in underlying code. The risk and repercussions associated with these vulnerabilities can range from simple memory overflow issues to critical security breaches, such as those linked to access controls. As depicted in Figure 1, the severity of such threats underscores the need for efficient and effective vulnerability management solutions.

```
module locked_register( input [15:0] Data_in,
input clk, resetn, write, lock_status, debug_unlocked,
outputreg [15:0] Data_out ) ;
always @( posedge clk or negedge resetn ) begin
    if ( ~ resetn ) begin
        Data_out <= 16'h0000 ;
    end
    else if ( write &(~ lock_status | debug_unlocked )
         ) begin
        Data_out <= Data_in ;
    end
    else if (~write) begin
        Data_out <= Data_out ;
    end
end
endmodule
```

Figure 1: Locked Register Bug $L8$: debug signal overrides lock status signal. Written in Hardware Description Language Verilog.

Over the past decade, Automatic Program Repair (APR) tools that leverage deep learning models (Wu et al., 2023; Steenhoek et al., 2023) have emerged as potential solutions to address code vulnerabilities. These tools, while efficient for many tasks, often struggle with more complex, high-threat bugs. On the other hand, Large Language Models (LLMs) like GPT-3 and GPT-4 have recently gained traction in addressing these challenging vulnerabilities (Ahmad et al., 2023; Xia et al., 2023). Refer Figure 2 where a simple prompt "`Identify the bug in the code and suggest a fix`" to the ChatGPT model accurately localised the bug and suggested the correct fix. However, obtaining the desired fix from these models remains a challenge, as they are not yet cost-effective due to multiple API calls often required for each identified bug and they require intricate prompting to yield the desired results.

Our objective is to combine the strengths of APR tools and LLMs, providing a more comprehensive and cost-effective solution for detecting and rectifying software vulnerabilities.

We aim to introduce a dual-layered approach. Initially, we'll use APR tools to address basic bug detection and remediation. For more sophisticated issues, we'll transition to LLMs. Alongside this, we plan to delve into incorporating advanced code representation methodologies into APR tools to improve their efficiency, moving beyond the prevalent approach of interpreting code as mere linear sequences of string tokens. Specifically, we're intrigued by graph-based embedding techniques, which capture the semantic and structural intricacies of code, as highlighted in (Zhou et al., 2019; Dinella et al., 2020).

The success of our approach will be gauged by its efficiency in bug (i) detection and (ii) fix. By the project's end, we aim to enhance the accuracy and efficiency of vulnerability detection tools while reducing the cost associated with leveraging LLMs, and uncover novel code representation techniques that could further push the boundaries of what's possible in this domain.

2. Background

Diving deeper into the vulnerability landscape, the Common Weakness Enumerations (CWEs) list by (Corporation, 2022) provides invaluable insights. CWEs present a taxonomy that classifies vulnerabilities in hardware and software, providing a systematic taxonomy to understand and tackle them. While some vulnerabilities, like simple memory overflow, can be addressed using existing APR tools, others, particularly those concerning access controls, prove to be more daunting. Studying the comprehensive nature of CWEs would serve as a good starting point to begin our understanding of code vulnerabilities especially those pertaining to security.

3. Methods and Evaluation Plan

An early design of the solution would entail a Two-Tier Detection and Resolution System. **Tier-1**: Employ APR tools to scan and rectify easily detectable bugs. **Tier-2**: Deploy LLMs for unresolved bugs, optimizing prompt tech-

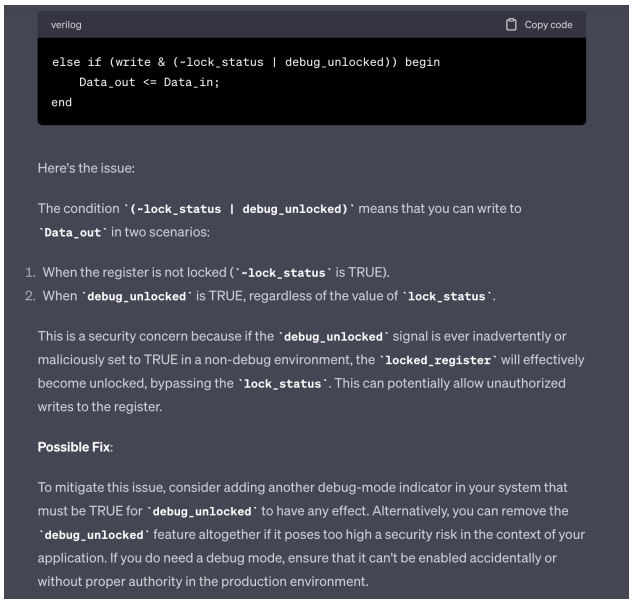

Figure 2: ChatGPT Sample Response correctly localising the vulnerability in $L8$ of Figure 1 and suggesting the fix to remove debug signal in the condition.

niques for precision and cost-effectiveness. Lastly, explore advanced embedding techniques that capture the semantic and structural essence of code, moving beyond its linear interpretation.

Finally, to evaluate the system we intend to use the following plan -

- **Efficiency Metrics**: Compare the number of detected vulnerabilities before and after the implementation of our system. The larger the difference, the more efficient our system.
- **Cost Analysis**: Monitor the number of API calls made to LLMs and calculate associated costs. A decrease in calls without compromising bug resolution would indicate success.
- **Code Representation**: Evaluate improved code representation techniques by assessing the model's understanding and accuracy when confronted with various vulnerabilities.

For the successful execution of our project, we would require:

- Access to a comprehensive dataset of software vulnerabilities like Vul4J (Bui et al., 2022), MSR (Fan et al., 2020), Devign (Zhou et al., 2019).
- Decent computational resources to test deep learning models and possible train them if required.
- Collaboration with domain experts for insights and validation.

4. Incremental Rollout of Project

- ☐ Begin by exploring datasets with vulnerabilities. Evaluate if they contain annotations for corresponding bug fixes and grade the severity of the bugs.
- ☐ Conduct a survey on existing small-scale language models that can detect, fix, or perform both tasks. Enumerate CWEs that pose the greatest challenges for detection and resolution.
- ☐ Use large-scale language models like GPT-3.5, which possess superior reasoning capabilities, to identify and rectify the previously identified bugs.
- ☐ Propose a comprehensive solution for vulnerability identification and repair.

References

Ahmad, B., Thakur, S., Tan, B., Karri, R., and Pearce, H. Fixing hardware security bugs with large language models. *arXiv preprint arXiv:2302.01215*, 2023.

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 *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 464–468, 2022.

Corporation, M. The mitre corporation. 2022. cwe - cwe-1194: Hardware design (4.1). 2022. URL https://cwe.mitre.org/data/definitions/1194.html.

Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Fan, J., Li, Y., Wang, S., and Nguyen, T. N. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pp. 508–512, 2020.

Steenhoek, B., Rahman, M. M., Jiles, R., and Le, W. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2237–2248. IEEE, 2023.

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. *arXiv preprint arXiv:2305.18607*, 2023.

Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.