Poster: Enhancing Symbolic Execution with LLMs for Vulnerability Detection

Muhammad Nabel Zaharudin
University of Texas at Arlington
muhammad.zaharudin@mavs.uta.edu

Muhammad Haziq Zuhaimi
University of Texas at Arlington
muhammad.zuhaimi@mavs.uta.edu

Faysal Hossain Shezan
University of Texas at Arlington
faysal.shezan@uta.edu

***Abstract*—Memory-related software vulnerabilities pose significant risks, allowing attackers to crash systems, execute unauthorized code, or steal sensitive data. Traditional symbolic execution tools like KLEE, effective in vulnerability analysis, face challenges like path explosion and scalability issues, particularly in large codebases. Conversely, Large Language Models (LLMs), skilled in parsing natural language and code, often struggle with pinpointing specific vulnerabilities. We propose a hybrid architecture that integrates KLEE's analytical precision with LLMs' pattern recognition capabilities. This approach utilizes LLMs to identify and refine critical code sections before symbolic execution by KLEE, enhancing accuracy and efficiency. Our method detected 27 out of 30 memory-related vulnerabilities, achieving 90% accuracy and reducing analysis time by 71.4%, from 3.5 hours to just 1 hour. This model not only improves efficiency in analyzing Linux kernel vulnerabilities but also offers potential for broader applications in diverse software environments.**

1. Introduction

Software vulnerabilities, especially memory-related bugs, pose a critical threat to the safety and security of our digital infrastructure. These flaws can open avenues for attackers to cause system crashes, execute unauthorized code, or steal sensitive data. The need for robust and efficient vulnerability discovery methods is thus paramount in today's software development landscape.

Symbolic execution tools like KLEE offer a powerful way to analyze software for potential vulnerabilities. However, they often suffer from path explosion – the exponential growth of possible execution paths – and scalability issues when dealing with large, complex codebases. Cadar et al. investigated that KLEE takes approximately 89 hours to successfully process over 141,000 lines of code [1]. On the other hand, LLMs excel in understanding natural language and code patterns, but their confidence in pinpointing specific vulnerabilities is less than ideal.

To address these challenges, we introduce our hybrid tool that leverages the strengths of both symbolic execution and LLMs. LLM is used to intelligently narrow down potentially problematic code sections within a large codebase and extract definitions of complex data structures. The extracted code fragments and data structure definitions are then fed into KLEE for focused, in-depth symbolic execution. This approach aims to mitigate path explosion, improve scalability, and increase the accuracy of vulnerability detection.

```
int afu_mmio_region_get_by_offset(struct dfl_feature_platform_data *pdata,
            u64 offset, u64 size, struct dfl_afu_mmio_region *pregion) {
    struct dfl_afu_mmio_region *region;
    struct dfl_afu *afu;
    int ret = 0;
    mutex_lock(&pdata->lock);
    afu = dfl_fpga_pdata_get_private(pdata);
    for_each_region(region, afu) {
        if (region->offset <= offset &&
            region->offset + region->size >= offset + size) {
            *pregion = *region;
            goto exit;
        }
    }
    ret = -EINVAL;
exit:
    mutex_unlock(&pdata->lock);
    return ret;
}
```

Figure 1: Successful LLM-Generated KLEE Code

Our hybrid approach, blending LLMs with KLEE, identified 27 out of 30 memory-related vulnerabilities, achieving a 90% accuracy. Despite challenges with complex integer overflows, it significantly improved CVE analysis efficiency, cutting down the computation time from 3.5 hours to just 1 hour—a 71.4% reduction. Our method not only boosts efficiency in Linux kernel vulnerabilities but also promises broader application in diverse software environments.

2. Background

For our research, we are utilizing KLEE as our symbolic execution engine to identify vulnerabilities in our code. KLEE requires that any given code is compiled to bytecode under the circumstance of the LLVM compilation framework. After compiling the program to bytecode, KLEE is able to execute and map instructions to variables as symbols for testing. KLEE would generate execution paths that would test these symbols for any errors and bugs. A user can declare which variable to store as a symbol and tests its symbolic states with function klee_make_symbolic() provided by the KLEE header. Based on the program being tested, the amount of test cases that are generated can vary.

3. Motivating Scenario

Our intuition is to utilize LLMs to automate and effectively reduce the effort for the KLEE by summarizing key areas of vulnerability in a large-scale program would overcome the challenges when handling large applications and avoiding path explosion problem. Figure 1 showcases

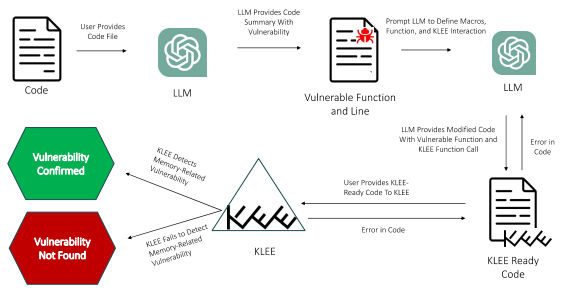

Figure 2: System Overview

that ChatGPT is able to generate correct KLEE code which pinpoints key symbols that pertain to the vulnerability in the given function. Capitalizing on ChatGPT-4's rapid response time, we would significantly reduce the expected runtime for KLEE when dealing with an extensive project.

4. Dataset

We collect 30 CVEs focused on memory-related vulnerabilities in C or C++ environments from the CVE database www.cvedetails.com. For each CVE, we locate the corresponding GitHub repositories containing both the vulnerable and patched code. By analyzing the differences, we identify the specific lines where vulnerabilities are present. These identified lines, along with the surrounding vulnerable code snippets, are then processed by our tool for further analysis.

5. System Design

In our study, we employ KLEE as a symbolic execution engine to detect vulnerabilities in code, generating execution paths that test for errors and bugs (Figure 2). We complement this with ChatGPT (V4-1106-Preview) and Gemini AI (V1.5) to leverage their enhanced capabilities over previous versions. For initial vulnerability detection, we used Gemini and for the later rounds we used ChatGPT. From our manual investigation, we found that this combination worked the best in our scenario. The research involves creating prompts that challenge the AI to identify general and specific vulnerabilities, including memory-related issues, in code snippets. Initial queries aim to detect any potential vulnerabilities, which may include excess or irrelevant data. For example, we provide the prompt 'What vulnerability exists in the following code?' to list any detected vulnerable lines in the given code snippet. Subsequent prompts refine this by pinpointing exact lines of code where vulnerabilities may exist, enabling us to assess the AI's ability to provide precise and actionable insights. For example, 'Can you find the line in the code snippet which might cause a vulnerability?', followed by 'Can you find the line in the code snippet which might cause a specific vulnerability?' with the specific vulnerability (e.g., memory-related) stated to narrow the focus to the desired type.

Furthermore, the AI modifies vulnerable code snippets to include necessary KLEE function calls for static analysis. If KLEE encounters errors due to complex data structures or unknown functions within these snippets, the AI is tasked with resolving these by modifying the code or defining missing elements. This iterative process allows KLEE to reattempt the analysis, enhancing its ability to successfully identify vulnerabilities or clarify its inability to do so, thus refining our understanding of both the AI's and KLEE's diagnostic capabilities in real-time coding scenarios.

Task	Component Used	Time
Potential Vulnerability Detection	LLM	5 minutes
KLEE Code Generation	LLM	15 minutes
Debugging	KLEE+LLM	30 minutes
KLEE Execution	KLEE	1 minute

TABLE 1: Task Completion Times (average).

6. Evaluation & Result

Our hybrid approach successfully identified 27 out of 30 memory-related vulnerabilities in selected CVEs (90% accuracy). Whereas, using only KLEE we were able to detect only 12 vulnerabilities. While effective, our tool has limitations when input files contain complex integer overflows that obscure data structures within the code. Despite this, we saw significant efficiency gains: our method reduced average CVE analysis time from 3.5 hours to 1 hour (71.4% reduction). In Table 1, we illustrate the time required for major This improvement stems from the LLM's ability to automate dependency declaration, macro definition, and symbolic value handling, tasks that were previously time-consuming and required substantial human effort.

The core principles behind our approach hold promise for broader applications beyond memory-related issues in the Linux kernel. The combination of an LLM's code comprehension capabilities with KLEE's symbolic execution offers a powerful foundation for vulnerability detection in diverse software and vulnerability types

7. Case Study

In Figure 3, we have illustrated a KLEE executable code that we have generated using our tool for CVE-2023-38427.

```
static __le32 deassemble_neg_contexts(struct ksmbd_conn *conn,
struct smb2_negotiate_req *req, int len_of_smb) {
    ksmbd_debug(SMB, "decoding %d negotiate contexts\n", neg_ctxt_cnt);
    if (len_of_smb <= offset) {
        ksmbd_debug(SMB, "Invalid response: negotiate context offset\n");
        return status;
    }
    len_of_ctxts = len_of_smb - offset;
    //...
    return status;
}
int main() {
    int len_of_smb;
    ...
    {klee_make_symbolic(&len_of_smb, sizeof(len_of_smb), "len_of_smb");}
    deassemble_neg_contexts(&conn, &req, len_of_smb);
    return 0;
}
```

Figure 3: KLEE Executable Code

References

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI* (2008).

Enhancing Symbolic Execution with LLMs for Vulnerability Detection

Muhammad Nabel Zaharudin, Muhammad Haziq Zuhaimi, Faysal Hossain Shezan
muhammadnabel45@gmail.com | haziqzuhaimi@ymail.com | faysal.shezan@uta.edu
University of Texas at Arlington – College of Engineering

Introduction

- Many open-source software projects are built with memory-unsafe programming languages.
- Memory-related vulnerabilities are a prevalent issue in software.
- Symbolic execution (e.g., KLEE) is a technique used to find code vulnerabilities.
- KLEE faces challenges with large codebases and path explosion.
- Our research introduces a hybrid solution combining Large Language Models (LLMs) and KLEE to detect memory vulnerabilities.

Motivation

- Identifying memory vulnerabilities poses a tough challenge for programmers and tools (e.g., KLEE) for large code base.
- According to research on KLEE's capabilities, Cadar et al. provided that KLEE roughly takes approximately ~89 hours to successfully process over 141,000 lines of code and provide effective test coverage.[1]
- LLMs automate and reduce effort for KLEE by summarizing vulnerable areas
- Our hybrid solution improves KLEE's usability and accessibility. .

```
int afu_mmio_region_get_by_offset(void *pdata, uint64_t offset, uint6
4_t size,  dfl_afu_mmio_region_t *pregion) {
   ...
   for_each_region(region, afu)
      if (region->offset <= offset &&
         region->offset + region->size >= offset + size){
         *pregion = *region;
         goto exit;  }
   ret = -1; // Symbolic return value
... }
int main() {
   // Symbolic inputs
   uint64_t offset, size;
   // Make symbolic
   klee_make_symbolic(&offset, sizeof(offset), "offset");
   klee_make_symbolic(&size, sizeof(size), "size");

         // Function call
   return afu_mmio_region_get_by_offset(&pdata, offset, size, &pregio
n);
 ...}
```

Figure 1: Successful LLM-Generated KLEE Code

System Design

- KLEE generates execution paths to test symbols for errors and bugs.
- We utilize ChatGPT (V4) and Gemini AI (V1.5) due to their improved capabilities.

LLM Prompts	Purpose
1. "What vulnerability exists in the following code"	LLM will list various possible vulnerabilities it can detect in the code snippet.
2. "Can you find the line in the code snippet which might cause a vulnerability?"	LLM will list possible lines that may cause vulnerability in the code based on the previous lists of vulnerabilities it could detect.
3. "Can you find the line in the code snippet which might cause a memory-related vulnerability?"	LLM will further narrow down the previous lists of lines to only lines that may cause a memory-related vulnerability.

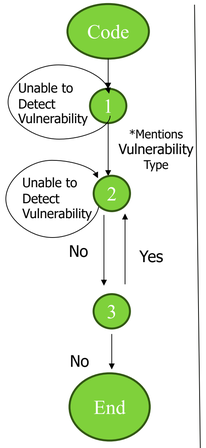

Figure 2: Series of prompts given to LLM to identify the line(s) of vulnerability

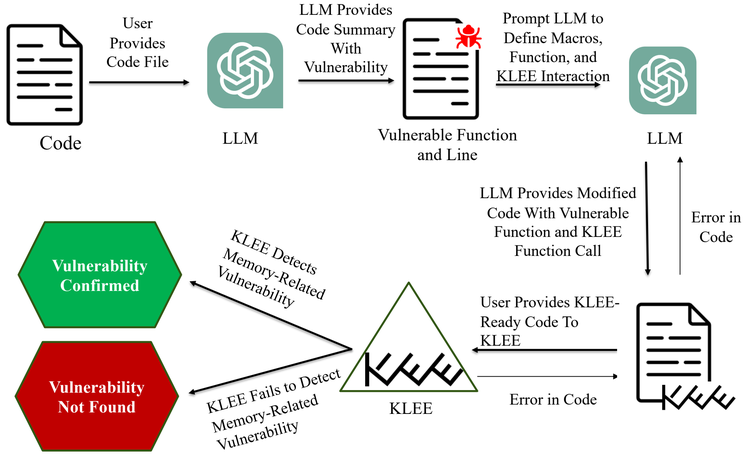

Figure 3: Conversation Between User and LLM

Case Study

Vulnerable Code

```
static __le32 deassemble_neg_contexts(struct
ksmbd_conn *conn,struct smb2_negotiate_req
*req,int len_of_smb)
{
  struct smb2_neg_context *pctx = (struct
smb2_neg_context*)req;
  int i = 0, len_of_ctxts;
  int offset = le32_to_cpu(req-
>NegotiateContextOffset);
  int neg_ctxt_cnt = le16_to_cpu(req-
>NegotiateContextCount);


if (len_of_smb <= offset) {  //vulnerable line
  ksmbd_debug(SMB, "Invalid response: negotiate
context          offset\n");
  return status;
}
  len_of_ctxts = len_of_smb - offset;
  ...
      return status;
}
```

KLEE Ready Code

```
static __le32 deassemble_neg_contexts(struct
ksmbd_conn *conn,struct smb2_negotiate_req *req,int
len_of_smb){
  ksmbd_debug(SMB, "decoding %d negotiate
contexts\n", neg_ctxt_cnt);
  if (len_of_smb <= offset) {
    ksmbd_debug(SMB, "Invalid response: negotiate
context offset\n");
    return status;
  }
  len_of_ctxts = len_of_smb - offset;
  //...
  return status;
}
int main() {
  int len_of_smb;
  ...
  klee_make_symbolic(&len_of_smb,sizeof(len_of_sm
b), "len_of_smb");
  // Call the function of interest
  deassemble_neg_contexts(&conn, &req,
len_of_smb);
  return 0;
}
```

Figure 6: LLM-Generated KLEE Test Code. LLM Modifies Code to be Compatible with KLEE

```
//…
ktest file : 'klee-last/test000001.ktest'
args       : ['klee_cve6.bc']
num objects: 1
object 0: name: 'len_of_smb'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
…//
//…
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_cve6.bc']
num objects: 1
object 0: name: 'len_of_smb'
object 0: size: 4
object 0: data: b'\xff\xff\xff\xff'
object 0: hex : 0xffffffff
object 0: int : -1
object 0: uint: 4294967295
object 0: text: ....
…//
```

Figure 7: KLEE Generated Test Cases for CVE-2023-38427 which Targets Vulnerabilites and Simultaneously Reports Them

Experiment Design and Evaluation

Task	Time
Vulnerability Detection	5 minute
KLEE Code Generation	15 minute
Debugging	30 minute
KLEE Execution	1 minute

Figure 5: Computational Overhead

- Our approach successfully detected 27 out of 30 CVEs, achieving **90% accuracy.**
- Using KLEE alone, we were able to detect 12 vulnerabilities.
- Our method reduced average CVE analysis time from 3.5 hours to 1 hour (a 71.4% reduction).

Conclusion

- The results demonstrate that the LLM is capable of identifying vulnerable code within a program.
- Our approach further streamlines the process, reducing average time by 71.4%.
- These results highlight the potential for automation in vulnerability detection.

Future Plans:

- Effectiveness of this approach to evaluate across a broader range of vulnerabilities.
- Thorough assessment required to understand potential performance implications of using larger and more complex datasets.
- The approach's potential for zero-day vulnerability detection

Acknowledgments

We want to thank the UTA Departmental REU program for their support. We also want to thank Microsoft Azure for providing Azure credits and the anonymous reviewer for their constructive feedback.

References

1.Cadar, C., Dunbar, D., & Engler, D. R. (2008, December). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (Vol. 8, pp. 209-224).