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, uint64_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, &pregion);
 ...}
```

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.

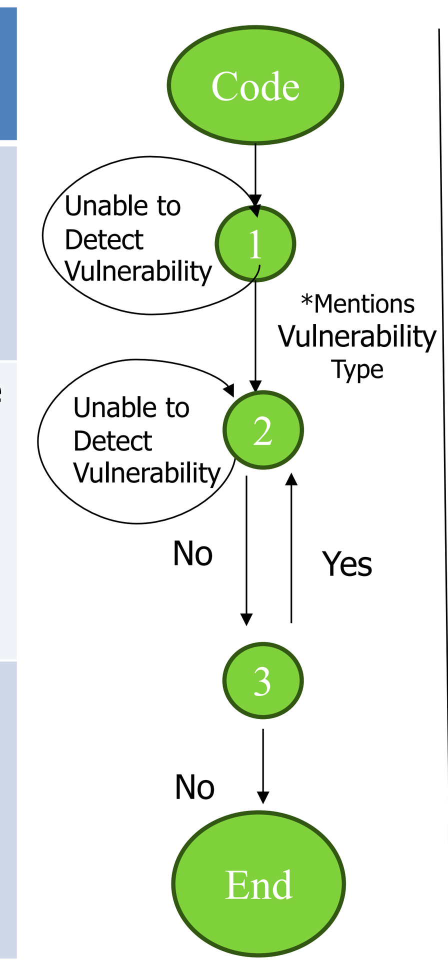

Figure 2: Series of prompts given to LLM to identify the line(s) of vulnerability

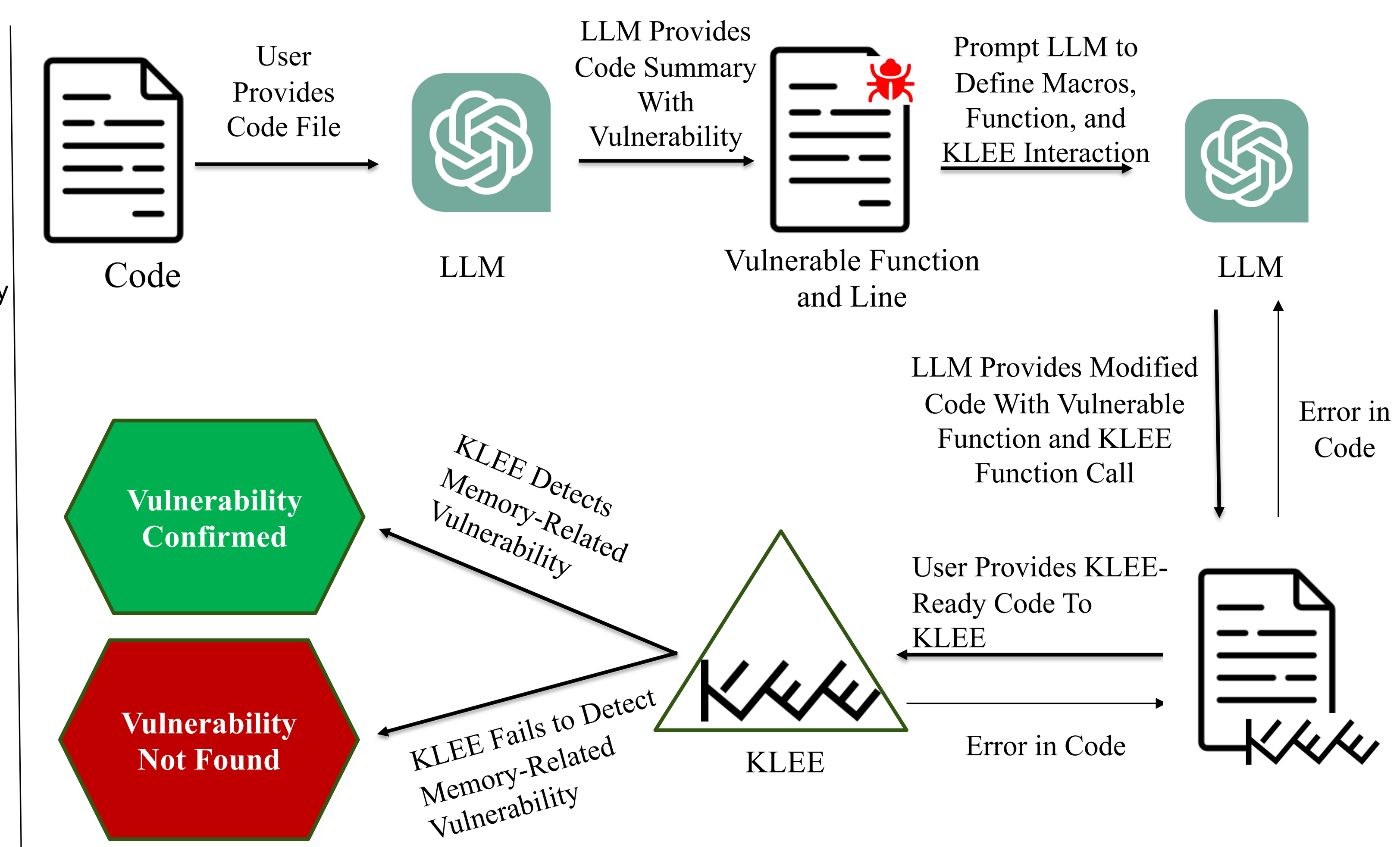

Figure 3: Conversation Between User and LLM

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_smb), "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).