Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach

aka: **The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models**

Haonan Li, Yu Hao, Yizhuo Zhai , Zhiyun Qian

UC Riverside

ABSTRACT:

- We investigate where and how ***Large Language Models (LLMs)*** can assist ***static analysis*** by asking appropriate questions. We target a specific bug-finding static analysis tool (UBITect) that produces a large number of *undecided cases* due to timeout/unknown APIs. We use LLM to handle these cases.
- Under 300 samples, we find our tool reach a precision of 50%, w/o missed bugs.
- With 1000 samples, we find 13 new bugs.

Background: UBITect

- UBITect is an analyzer and aims to find *Use-before-initialization* (UBI) bugs all over the Linux kernel.
- UBITect uses *symbolic execution* to reduce false positives, but suffers from timeout/out of memory and leaves many undecided cases.
- Our proposed tool, "LLift", takes these undecided cases and analyze them under LLM.

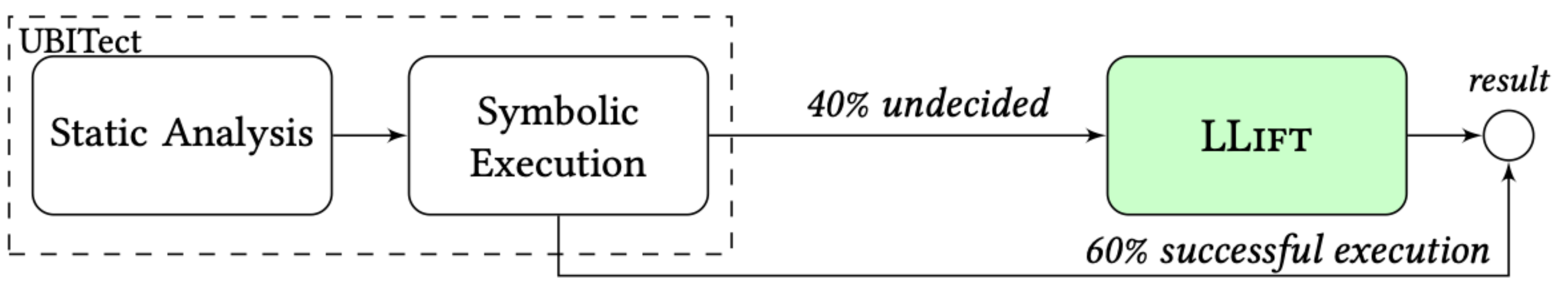

Figure 1: The high-level workflow of LLift. Take undecided cases by UBITect and determine whether these cases are real bugs.

Motivation:

```
static int libcfs_ip_str2addr(...){
  unsigned int a, b, c, d;
  if (sscanf(str, "%u.%u.%u.%u%n",
          &a, &b, &c, &d, &n) >= 4 && ...){
          // use of a, b, c, d
    }
}

int sscanf(const char *buf,  const char *fmt, ...){
  va_start(args, fmt);
  i = vsscanf(buf, fmt, args);
  va_end(args);
}
```

Figure 2: Code snippet of sscanf and its usecase, derived from Linux kernel

- In the sscanf case, UBITect lacks return value sensitivity (ret >=4 check). It simply estimates that all parameters *"may" be left uninitialized.*
- LLM can resolve this case because:
  - It is familiar with function \`sscanf\`, knowing the meaning of *return value*.
  - With the return value>=4, it can infer the \`sscanf\` must initialize first four parameters.

Paper Link:

https://dl.acm.org/doi/10.1145/3649828

Homepage:

https://sites.google.com/view/llift-open

Return Value Check (Post-Constraint):

Table 1: Two types of post-constraints: check before use, failure check

Check Before Use	Failure Check

Type A:
`if (sscanf(...) >= 4) {`
`  use(a, b, c, d);`
`}`
	Type B:
	`err = func(&a);`
	`if (err) { return/break/goto; }`
	`use(a)`

Type A':
`switch(ret=func(&a)){`
`  case some_irrelevant_case:`
`    do_something(...);`
`    break;`
`  case critical_case:`
`    use(a);`
`}`
	Type B':
	`while(func(&a)){`
	`    do_something(...);`
	`}`
	`use(a);`

- More generally, not all paths are important for UBI, only "use" paths are essential.
- By focusing on return value checks (post-constraint), we can prune out many paths.

Workflow:

LLift prompts LLM to:

- Identify the "initializer"
- Extract the post-constraints
- Analyze the initializer with post-constraints

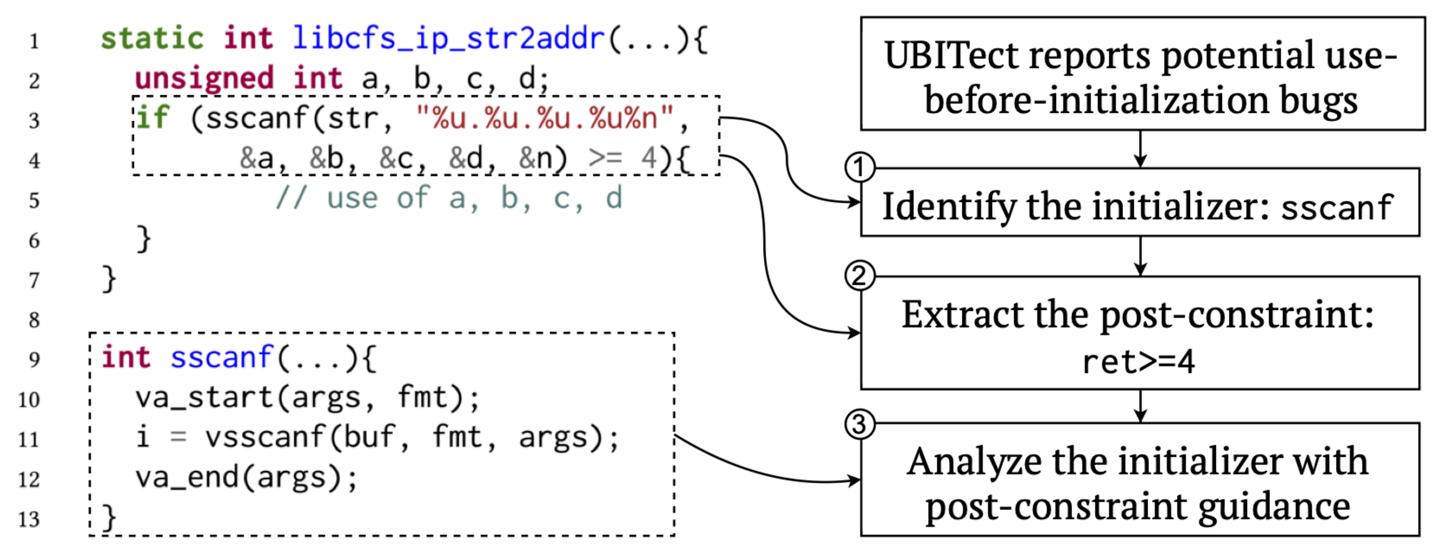

Figure 3: Example run of LLift.

Prompt Design: Progressive Prompt

- Linux kernel has a very large codebase, functions are often called across files.
- ***Progressive prompt***: dynamically provides function definition when LLM *needs*.

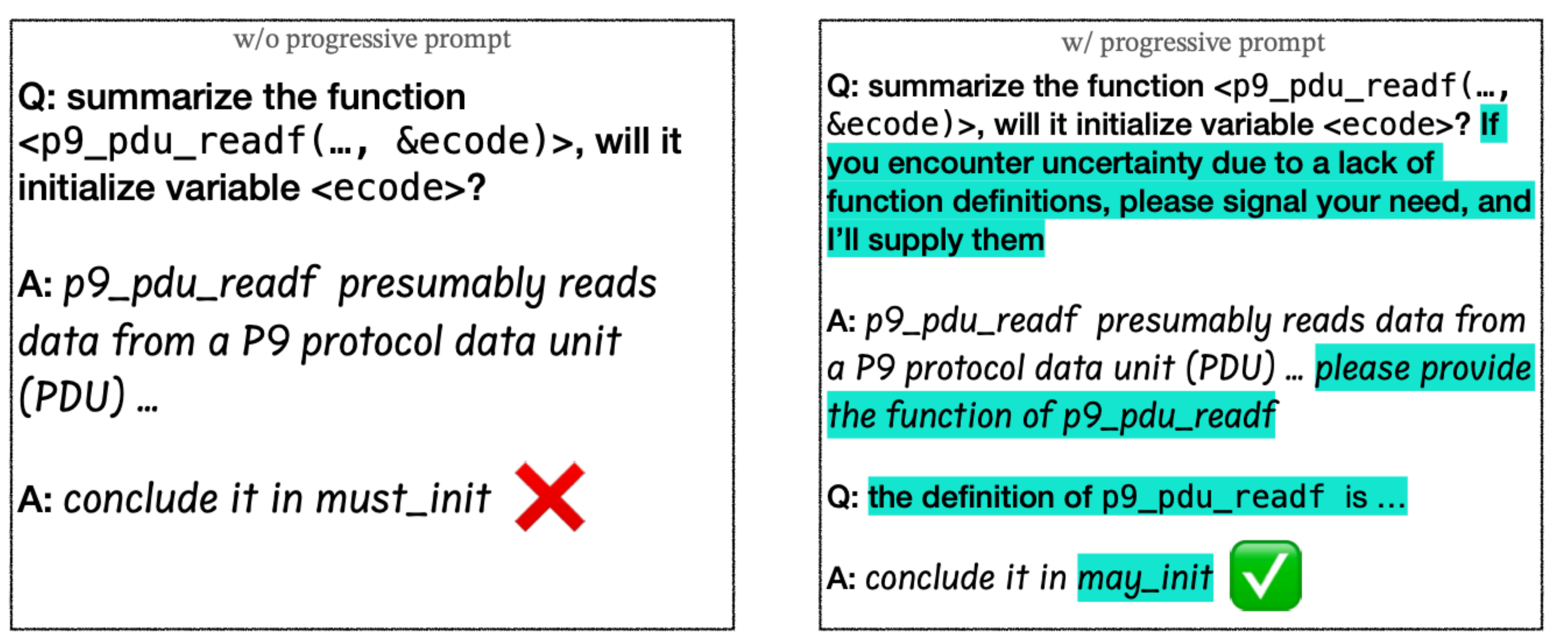

Figure 4: A demonstration of progressive prompt. The prompts and responses are simplified.

Result:

- 13 new bugs.
- 50% precision (on undecided cases), not found false negatives yet.
- All design components are useful: simple prompt has 12% precision and 15% recall.