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

HAONAN LI, University of California, Riverside, USA
YU HAO, University of California, Riverside, USA
YIZHUO ZHAI, University of California, Riverside, USA
ZHIYUN QIAN, University of California, Riverside, USA

While static analysis is instrumental in uncovering software bugs, its precision in analyzing large and intricate codebases remains challenging. The emerging prowess of *Large Language Models* (LLMs) offers a promising avenue to address these complexities. In this paper, we present LLift, a pioneering framework that synergizes static analysis and LLMs, with a spotlight on identifying *Use Before Initialization* (UBI) bugs within the Linux kernel. Drawing from our insights into variable usage conventions in Linux, we enhance path analysis using post-constraint guidance. This approach, combined with our methodically crafted procedures, empowers LLift to adeptly handle the challenges of bug-specific modeling, extensive codebases, and the unpredictable nature of LLMs. Our real-world evaluations identified four previously undiscovered UBI bugs in the mainstream Linux kernel, which the Linux community has acknowledged. This study reaffirms the potential of marrying static program analysis with LLMs, setting a compelling direction for future research in this area.

CCS Concepts: • **Security and privacy** → **Systems security**; • **Computing methodologies** → *Natural language processing*; • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: Static analysis, bug detection, large language model

ACM Reference Format:
Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. https://doi.org/10.1145/3649828

1 INTRODUCTION

Static analysis has long stood as an important tool to understand program behaviors and improve code quality, reliability, and security. Especially in the realm of bug discovery, it offers a proactive mechanism to unearth bugs before the code is shipped to production. Yet, the complexity of modern software, exemplified by massive codebases like the Linux kernel, strains the limits of static analysis. At the heart of this lies a fundamental trade-off: precision versus scalability [Gosain and Sharma 2015; Park et al. 2022].

When applied to bug detection, precise static analyses, such as path-sensitive ones, can effectively distinguish between actual bugs and false alarms. However, they fall short when faced with complex programs. Conversely, scalable analyses, designed for giving quick but over-approximate answers, suffer from a large number of false positives, clouding their utility and adoption in practice. UBITect highlights this contrast [Zhai et al. 2020]. Tailored for detecting *Use Before Initialization* (UBI) bugs

Authors' addresses: Haonan Li, University of California, Riverside, Riverside, USA, hli333@ucr.edu; Yu Hao, University of California, Riverside, Riverside, USA, yhao016@ucr.edu; Yizhuo Zhai, University of California, Riverside, Riverside, USA, yzhai003@ucr.edu; Zhiyun Qian, University of California, Riverside, Riverside, USA, zhiyunq@cs.ucr.edu.

ACM 2475-1421/2024/4-ART111
https://doi.org/10.1145/3649828

in the Linux kernel, UBITect adopts a two-tiered bug detection strategy: a path-insensitive static analysis for scalability and a follow-up symbolic execution (path-sensitive by definition) phase for precision. Yet, even with this two-tier design, approximately 40% of the potential bugs reported in the static analysis phase failed to be completed due to timeouts or memory limits in the symbolic execution step.

The dilemma is clear: ignoring potential bug reports might mean missing genuine issues, but forwarding them all to developers could swamp them with numerous false positives. The rise of *Large Language Model*s (LLMs) provides an encouraging solution to address this challenge. Their proven ability to understand, generate, and even debug code offers an innovative approach to tackle the inherent challenges of static analysis, as evidenced by recent studies [Chen et al. 2021; Copilot 2023; LangChain 2023]. However, it's not without its caveats. LLMs, while impressive, can sometimes "hallucinate" — creating fictitious facts or misinterpreting code. Additionally, their inherent randomness might lead to inconsistent analyses, and their limited context windows might restrict the depth of their evaluations [Ji et al. 2023].

In this light, this paper presents LLift, an automated framework enhancing static analysis in bug detection with LLMs. Firstly, we introduce the concept of *Post-Constraint Guided Path Analysis*, an optimization under path-sensitive analysis. By focusing on the constraints that trigger the bugs, this method can reduce the exploration paths and make the analysis of intricate vulnerabilities more precise. Secondly, we address the hurdles and potential solutions of integrating LLMs into static analysis for bug identification. Our pioneering framework, LLift, exemplifies the seamless integration of classical static analysis and the prowess of LLMs.

We evaluate LLift primarily on the Linux kernel. Taking the undecided UBI warnings (40% for the Linux kernel) in UBTect, we identified four new bugs in the mainstream Linux kernel. These discoveries have been acknowledged by the Linux community. By fusing an optimized path analysis with the discerning capabilities of LLMs, our goal is to propel static analysis into a new era of bug identification. This paper documents this exploration, detailing our strategies and insights from practical implementations.

We summarize our contributions as follows:

- **New Opportunities.** We introduce a novel approach to static analysis that enhances its capability for bug detection by introducing LLMs. To the best of our knowledge, we are the first to demonstrate how to apply LLMs to address the limitations of static analysis and enhance its bug finding capabilities.
- **Post-Constraint Guided Path Analysis.** We leverage post-constraint guided path pruning in practical bug detection with large-scale codebases. It can effectively enhance the capabilities of static analysis in path-sensitive analysis.
- **Methodologies in Leveraging LLMs.** We develop LLift, an innovative and fully automated framework. LLift employs several prompt strategies to engage with LLMs, obtaining accurate and reliable responses.
- **Results.** We rigorously investigate LLift by analyzing nearly 300 undecided cases from UBITect, resulting in a reasonable precision rate (50%), and no missing bugs were found. Furthermore, LLift reveals 13 undiscovered bugs from UBITect with extensive tests on 1000 cases, and four are already confirmed as real bugs with the Linux community.

2 BACKGROUND & MOTIVATION

2.1 UBITect and Motivating Example

UBITect is a state-of-the-art static analysis solution aiming at finding *Use Before Initialization* (UBI) bugs in the Linux kernel [Zhai et al. 2020]. It employs a two-stage pipeline whereas the first

```
1   static int libcfs_ip_str2addr(...){
2     unsigned int a, b, c, d;
3     if (sscanf(str, "%u.%u.%u.%u%n", &a, &b,
    ↪  &c, &d, &n) >= 4){
4         // use of a, b, c, d
5     }
6   }
7   int sscanf(...){
8     va_list args;
9     int i;
10    va_start(args, fmt);
11    i = vsscanf(buf, fmt, args);
12    va_end(args);
13
14    return i;
15  }
```

```
16  int vsscanf(...){
17    const char *str = buf;
18    ...
19    while (*fmt) {
20      switch (*fmt++) {
21        case 'c': {
22          char *s = (char *)va_arg(args, char*);
23          ...
24          do { *s++ = *str++; }
25          while (...*str); num++;
26        }
27      }
28      ...
29      num++;
30          ...
31    }
32    return num;
33  }
```

Fig. 1. Code snippet of sscanf and its usecase. va_args and the unbounded loop in vsscanf make it difficult to analyze.

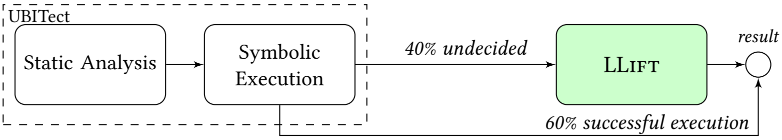


Fig. 2. The high-level workflow of LLift. Start with the undecided cases by UBITect and determine whether these potential bugs are true or false.

stage employs a flow-sensitive but path-insensitive static analysis. By design, this stage aims for scalability and sacrifices precision, producing a significant number of potential bugs (*i.e.,* ~140k), the vast majority of which are false alarms. The static analysis is imprecise primarily due to its *lack of path sensitivity.*

Path Sensitivity. A path-sensitive analysis can distinguish different paths by considering their path constraints. In the Linux kernel, the *conditional initialization* is a common practice, for example in the code snippet shown in Fig 1, the *check* at Line 3 ensures that a, b, c, and d must be initialized within the code block under this check (Line 4). However, the path-insensitive analysis cannot distinguish the different path constraints (under the check or not). Hence, UBITect would claim a potential bugs of *use-before-initialization* at Line 4 because it only considers the function sscanf() "*may initialize*" parameters a, b, c, and d at Line 3.

UBITect tries to solve this insensitivity by adding a second stage of symbolic execution that reduces false alarms by filtering their infeasible paths. However, 40% of the reported bugs are discarded (including this sscanf case) due to the threshold of time (*i.e.,* 10 minutes) or memory (*i.e.,* 2 GB) during the symbolic execution. As a result, despite the symbolic execution can filter some false positives, it could potentially missing genuine bugs because of the scalability issue of symbolic execution.

2.2 General Challenges of Static Analysis in Practices

Ostensibly, the false alarms on the `sscanf()` case are due to imperfections in the design and implementation of UBITect; however, we believe that the dilemma also represents the flaws that generally appear in program analysis. We summarize two flaws as follows:

Inherent Knowledge Boundary. Developers need to model specific functions or language features. Otherwise, they influence the correctness of the results. For compiler built-in functions, *e.g.,* `va_start()` in our motivating example, their definitions are simply not available. Beyond this example, an array of other scenarios are particularly prevalent in the Linux kernel. These situations include *assembly code*, *hardware behaviors*, *callback functions*, and *concurrency*.

In practice, the knowledge boundary problem can be solved with human modeling. For example, SVF [Sui and Xue 2016] manually identifies and models hundreds of common APIs in its analysis. However, in practical terms, it is often time-consuming to discover and model all these cases because they can be highly dependent on the analysis target, especially for a large and evolving codebase such as the Linux kernel. This limitation often compromises the effectiveness of static analysis, leaving it less precise and comprehensive than desired.

Exhaustive Path Exploration. Correctly handling cases like `sscanf()` is required to consider all its possible behaviors associated with the return value. However, in this case, as Figure 1 shown, the return value (num) of `sscanf()` is accumulated with two unbounded loops `while(*fmt)` and `while(*str)` (`str` is `buf`). Considering the exponential explorations for all possible paths, it becomes highly challenging to infer precisely which arguments will be initialized. In UBItect, it leads to the timeout in symbolic execution and, therefore, fails to confirm this potential bug.

In general, Existing path-sensitive static analysis (and symbolic execution) techniques operate under a methodical but exhaustive paradigm, exploring all potential execution paths through the codebase. While this approach is theoretically comprehensive, it often leads to path explosion.

2.3 Capability of LLMs

Recent advances in *Large Language Model*s (LLMs) [OpenAI (2023) 2023b] offer a promising alternative to summarizing code behaviors [Ouyang et al. 2022] in a flexible way which "sidestep" the aforementioned challenges. LLMs are trained with extensive datasets, including natural language and source code [Ouyang et al. 2022], they can effectively work with complicated code snippets and produce an *intuitive comprehension*; for example, they can summarize loop invariants that are hard to perform using traditional program analysis methods [Pei et al. 2023a]. Similarly, we observe that LLMs can recognize path conditions and control flow constructs within the provided code, allowing them to reason about different execution paths (even in loops). While traditional static analysis methods provide formal and exhaustive analysis, LLMs can present a more intuitive, human-like comprehension. Therefore, when traditional static analysis methods fall short, *e.g.,* as in the 40% undecided cases in UBITect, we resort to LLMs, which can complement existing methods.

3 PROBLEM FORMULATION

In this section, we first define the UBI bug and provide the general idea of LLift in §3.1. Then, we demonstrate our observation and optimization of path pruning in §3.2. Lastly, we propose a conceptual workflow of LLift in §3.3.

3.1 System Definition

3.1.1 Use-Before-Initialization. A *Use Before Initialization* (UBI) bug refers to the erroneous scenario where a variable v is accessed or involved in any operation prior to its correct initialization. Let:

- v is a local variable.
- F is the function that declares v.
- U signifies a use operation involving v.
- I represents an initializer that can initialize v.

Then v is *used before initialization* at U, if there exists a potential execution path that makes the use of v, *i.e.,* U ahead of all its initializers I. Note we consider both U and I as function invocations within F. Figure 3 demonstrates a simplified UBI bug. In this case, the variable v may be used (Line 5) before its initialization (Line 4) if the constraint is not satisfied (Line 3).

```
1  int F(){
2      int v; // declare of  v
3      if (constraint)
4          init(&v); // initializer of v
5      use(v); // use of v
6  }
```

Fig. 3. A typical example of a UBI bug.

3.1.2 Static Analysis Report (SAR). We assume an imprecise static analysis is applied to detect potential UBI bugs (*i.e.,* has false positives). Each potential bug is accompanied by an *Static Analysis Report* (SAR), which is a tuple as follows (symbols are defined above):

$$\text{SAR} = \langle v, U, F \rangle \tag{1}$$

3.1.3 Initializer and its effects. As mentioned, an initializer I is a function invoked within F that can initialize a variable v. More generally, an initializer can initialize more than a single variable. For example, for a function R with $Vars = \{v_0, v_1\}$, if R initializes v_0 in all cases (*must_init*) and v_1 under some conditions (*may_init*), the results of the initialization of variables can be expressed as $\text{FuncInit}(R) = \{v_0 \mapsto \textit{must\_init}, v_1 \mapsto \textit{may\_init}\}$. We could also say $\text{FuncInit}(R)[v_0] = \textit{must\_init}$.

Namely, we define the initialization of variables FuncInit with variables $Vars$ in an initializer as:

$$\text{FuncInit} = Vars \mapsto \{\textit{must\_init, may\_init}\} \tag{2}$$

In general, LLift takes the SAR as its input, finds possible initializers, analyzes the FuncInit of these initializers, and concludes whether the SAR indicates a real bug.

3.2 Post-Constraint Guided Path Analysis

The intuition of the post-constraint guided path analysis is to consider the initializer's return value (or any forms of ***outcome***) and the path constraint of the use of the suspicious variable (which we refer to as ***post-constraint***). By considering these constraints, we can usually get postconditions with fewer cases of initializers, and therefore, get a more precise bug report than UBITect.

3.2.1 Outcome and Postcondition. We consider the ***outcome*** $\mathcal{O}$ of initializers. Then, in the context of UBI detection, we define the ***postcondition*** $\mathcal{P}$ of an initializer as: $\mathcal{P} = \mathcal{O} \times \text{FuncInit}$. For example, the `sscanf(...)` case in Figure 1, its postcondition can be expressed as:

$$\begin{aligned}
\mathcal{P}_1 &: \{ret \mapsto 0, [\![\textit{must\_init}]\!] = \emptyset\} \\
\mathcal{P}_2 &: \{ret \mapsto 1, [\![\textit{must\_init}]\!] = \{a\}\} \\
\mathcal{P}_3 &: \{ret \mapsto 2, [\![\textit{must\_init}]\!] = \{a, b\}\} \\
\mathcal{P}_4 &: \{ret \mapsto 3, [\![\textit{must\_init}]\!] = \{a, b, c\}\} \\
\mathcal{P}_5 &: \{ret \mapsto 4, [\![\textit{must\_init}]\!] = \{a, b, c, d\}\} \\
\mathcal{P}_6 &: \{ret \mapsto 5, [\![\textit{must\_init}]\!] = \{a, b, c, d, n\}\}
\end{aligned}$$

Here, the $\mathcal{P}_1 - \mathcal{P}_6$ represent different possible postconditions of the call of `sscanf()`. $[\![must\_init]\!]$ is the set of all variables that must be initialized.

In the context of UBI detection, we notice that not all possibilities are worth noting; instead, only the outcomes making U ***reachable*** are required for UBI detection. We call the constraint that makes the *use* reachable ***post-constraint*** C_{post} [Huang 2007]. The *qualified postcondition*, $\mathcal{P}_{qual}$, is a subset of $\mathcal{P}$ refined by C_{post}:

$$\mathcal{P}_{qual} = \mathcal{P}|_{C_{post}}$$

For the `sscanf()` case, we have the post-constraint as C_{post} = ret ≥ 4; therefore, the qualified postcondition would be $\mathcal{P}_5 \wedge \mathcal{P}_6$, which ensures that variables a, b, c, and d must be initialized. Therefore, we have $\forall v \in \{a, b, c, d\}$, $\text{FuncInit}[v] = must\_init$, and no UBI bug happens in this case.

```
void I(int& v){
  if (c_post){
    //path1
    *v = 0;
  }
  else {
    //path2
  }
}
void F(){
    int v;
    I(v);
    if(c_post)
        use(v);
}
```

Fig. 4. An example of direct application. The $path_2$ can be pruned.

3.2.2 Post-Constraint Guided Analysis. Given the constraints of the use of the suspicious variable C_{post}, it is possible to prune paths that inherently do not meet those expectations when analyzing the initializer. Namely, we take the post-constraint in advance and enable the subsequent analysis to obtain the *qualified* postcondition of the initializer directly.

Specifically, we categorize the post-constraint guided analysis into two scenarios, ***direct application*** and ***outcome conflict***, in applying this optimization.

Direct Application. For example in Figure 4, we have C_{post} = `c_post` that makes the `use(v)` reachable, we then can conclude that $path_2$ can be pruned because its path constraint ϕ_2 = ¬`c_post` conflicts with C_{post}.

More formally, for a given C_{post} that is extracted from an SAR, we consider it with the path constraint in the initializer. For each $path_i$ in the initializer, the $path$ can be pruned if $\phi_{path} \wedge C_{post}$ is ***not*** satisfiable.

Outcome Conflict. For example in Figure 5, the C_{post} requires the return value of the initializer to be 0 ($ret\_value = 0$), the outcome of $path_2$ will cause ret_value to be -1, which conflicts with the post-constraint. This way, $path_2$ can be pruned.

```
int I(){
  if (...){ //path1
    return 0;
  }
  else{ //path2
    return -1;
  }
}
void F(){
    int v;
    int ret = I(v);
    if (ret == 0){
        use(v);
    }
}
```

Fig. 5. An example of outcome conflicts. The $path_2$ can be pruned.

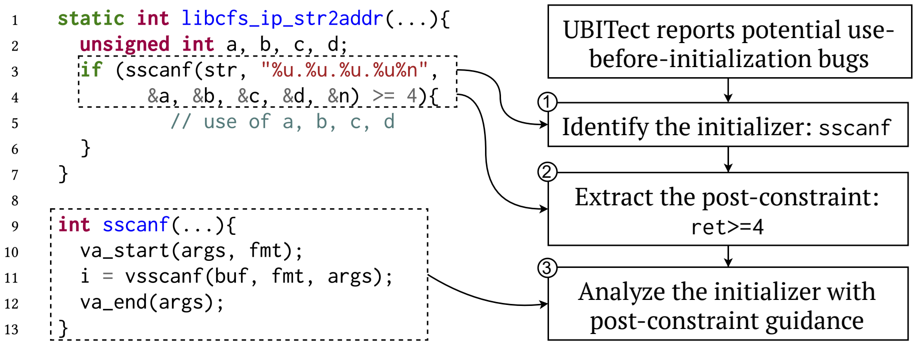
More formally, let $O(path)$ denote the set of all outcomes or effects produced by the $path$. Then the $path$ can be pruned if we can find an outcome $o \in O(path)$ that makes $o \wedge \neg C_{post}$ satisfiable.

The post-constraint guided analysis enables efficient path pruning in the analysis of the initializer. We leverage LLM to implement the analysis, and we show the details in §4.3.

3.3 Conceptual Workflow

Given a bug report (SAR) containing a suspicious variable v, its usage U, and the function F , the workflow Φ is as follows:

(1) $\Phi_1(\text{SAR}) \longrightarrow \{I\}$: Identify all potential initializers for v from the bug report.
(2) $\Phi_2(\text{SAR}, I) \longrightarrow C_{post}$: Extract the C_{post} from the bug report for each possible I.
(3) $\Phi_3(\text{SAR}, \{I, C_{post}\}) \longrightarrow \text{InitStatus}(v)$: Summarize the initialization status for variable v after all possible initializers, *i.e.*, $\bigcup_I \text{FuncInit}(I)[v]$. For v, if there exists any I that must initialize v, then the $\text{InitStatus}(v)$=*must_init*. with respect to their corresponding C_{post}.



```
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(...){
  va_start(args, fmt);
  i = vsscanf(buf, fmt, args);
  va_end(args);
}
```

UBITect reports potential use-before-initialization bugs
①
Identify the initializer: sscanf
②
Extract the post-constraint: ret>=4
③
Analyze the initializer with post-constraint guidance

Fig. 6. Example run of LLift. For each potential bug, LLift ① (Φ_1) identifies its initializer, ② (Φ_2) extracts the post-constraints of the initializer, and ③ (Φ_3) analyzes the behavior of the initializer with the post-constraints via LLM.

3.3.1 Decision Policy. The decision policy Δ is defined as:

$$\Delta(\text{InitStatus}(v) = \textit{must\_init}) : \text{non-bug}$$
$$\Delta(\text{InitStatus}(v) \neq \textit{must\_init}) : \text{potential bug}$$

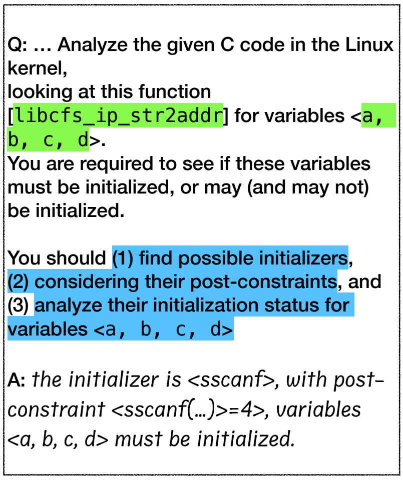
This policy adopts a conservative approach by treating all variables *not* marked as *must_init* as potential vulnerabilities. And it is worth noting that this policy may introduce some false positives. For example, it might over-approximate the initialization status due to the absence of path constraints outside F.

If Δ says the SAR is a potential bug, it considers this case from the static analysis as a true bug; otherwise, it is filtered. For single initializer cases (*i.e.,* only one possible initializer can be found), we can also directly see the result from FuncInit(I).

For multiple initializers with *may_init*, we need to consider the conditions of initializing v. This would theoretically require a path-sensitive analyzer to output the conditions under which I initializes v if it is *may_init*. Instead of dealing with these conditions precisely, we simply summarize them as *may_init*. The lack of considerations of (pre)conditions for multiple initializers could be one of the reasons for false positives, even though we do not find such cases in our experiment.

4 LLM-BASED PROGRAM ANALYSIS

We introduced a conceptual workflow in Section §3.3. Elaborating on that foundation, Figure 7 showcases a simple LLM-based analysis for our motivating example. We can see that it can successfully infer all variables of `a, b, c, d` must be initialized.

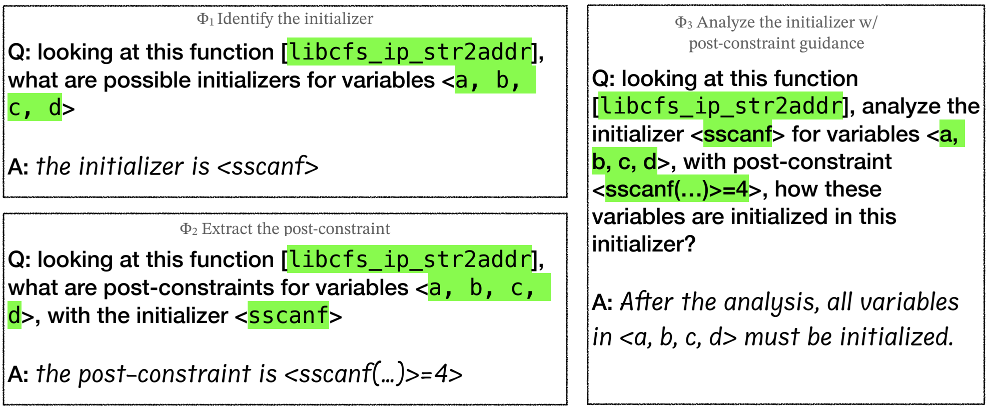


Q: ... Analyze the given C code in the Linux kernel,
looking at this function [`libcfs_ip_str2addr`] for variables <`a, b, c, d`>.
You are required to see if these variables must be initialized, or may (and may not) be initialized.

You should (1) find possible initializers, (2) considering their post-constraints, and (3) analyze their initialization status for variables <`a, b, c, d`>

A: *the initializer is <sscanf>, with post-constraint <sscanf(...)>=4>, variables <a, b, c, d> must be initialized.*

Fig. 7. Simple prompt design, following three stages in our workflow in §3.3. The case-specific prompts are highlighted in green. <> indicates the variables and functions, and the [] represent the function definition in source code.



Φ_1 Identify the initializer

Q: looking at this function [`libcfs_ip_str2addr`], what are possible initializers for variables <`a, b, c, d`>

A: *the initializer is <sscanf>*

Φ_2 Extract the post-constraint

Q: looking at this function [`libcfs_ip_str2addr`], what are post-constraints for variables <`a, b, c, d`>, with the initializer <`sscanf`>

A: *the post-constraint is <sscanf(...)>=4>*

Φ_3 Analyze the initializer w/ post-constraint guidance

Q: looking at this function [`libcfs_ip_str2addr`], analyze the initializer <sscanf> for variables <a, b, c, d>, with post-constraint <sscanf(...)>=4>, how these variables are initialized in this initializer?

A: *After the analysis, all variables in <a, b, c, d> must be initialized.*

Fig. 8. A three-step LLM-based analysis, following three stages in our workflow in §3.3. The case-specific prompts are highlighted in green. <> highlight the previous results, and the `[libcfs_ip_str2addr]` is a placeholder for the caller function definition of `libcfs_ip_str2addr` in the source code. The prompts and responses are simplified.

Yet, this simple prompt design is not effective in reality, especially for more complex cases than `sscanf()`, even with the advanced knowledge and analytical capabilities of cutting-edge LLMs. We also test this prompt in §6.3 as "simple prompt".

4.1 Prompt Overview

Figure 8 illustrates the overall process of a three-step LLM-based analysis. This design is aligned with the description in Figure 6. In general, this design decomposes the entire workflow into three pieces. Responses from the previous steps pass to the next turns, making each piece become smaller and more manageable for LLMs.

We also add more details at each step, shown in Figure 8. Because we also face a series of challenges in practice, especially for program analysis on the Linux kernel. We present these challenges and our solutions in the following parts of this section.

4.2 Design Challenges and Solutions

It is non-trivial to prompt LLMs effectively [Shieh 2023; Zhao et al. 2023]. During the design of LLift, we meet the following challenges and propose solutions correspondingly.

C1. Limited Understanding of Post-Constraint. Despite the latest LLMs (*e.g.,* GPT-4) being able to explain the definition of post-constraint and apply them in simple scenarios, their capacity to identify bugs, in general, is limited [OpenAI (2023) 2023b]. Without more explanation and guidance, LLMs would often ignore important post-constraints and identify bugs incorrectly. This is because of their limited training in program comprehension and bug detection.

C2. Token Limitations. It is known that *Large Language Models* (LLMs) have token limitations. For example, GPT-3.5 supports 16k tokens and GPT-4 supports 32k tokens [OpenAI (2023) 2023a]. The token limitation is a fundamental challenge for all transformer-based architectures, such as GPT series, due to the complexity of $O(N^2)$ in computing attention [Vaswani et al. 2017]. This restricts the provision of multiple function contents for our tasks.

C3. Unreliable and Inconsistent Response. LLMs are known to result in unreliable and inconsistent responses due to *hallucination* and *stochasticity* [Zhao et al. 2023]. Stochasticity refers to the inherent unpredictability in the model's outputs [Vaswani et al. 2017]; and the hallucination refers to LLMs generating nonsensical or unfaithful responses [Ji et al. 2023; Zheng et al. 2023]. By design, the stochasticity can be mitigated with lower *temperature*, a hyperparameter controlling the degree of randomness in outputs [Salamone 2021]; however, reducing temperature may impair the model's exploring ability [Xu et al. 2022] and, therefore may miss real vulnerabilities.

Facing these challenges, we propose the following design components to solve them:

- To tackle challenge **C1** (Post-Constraint), we propose to encode ***(D#1) Post-Constraint Guided Path Analysis.*** where we teach LLMs by examples of post-constraint, including code patterns. This is also a well-known strategy in LLM research, *i.e.,* few-shot in-context learning [Song et al. 2023]. This approach enables LLMs to learn from a small number of demonstrative examples, assimilate the underlying patterns, and apply this understanding to incorporate post-constraint guidance in our analysis.
- To tackle challenge **C2** (Token Limitation), We employ two strategies: ***(D#2) Progressive Prompt.*** Instead of copying a large number of function bodies (*i.e.,* subroutines), we only provide function details on demand, *i.e.,* when LLMs are not able to conduct a result immediately. ***(D#3) Task Decomposition.*** We break down the problem into sub-problems that can be solved easily. Figure 8 illustrates one aspect of task decomposition. Each step (Φ) is further split into multiple prompt-response pairs (we refer to ***turns***) in practice.
- To tackle challenge **C3** (Unreliable Response), we employ the following strategies: ***(D#4) Self-Validation.*** We ask LLMs to review and correct their previous responses. This helps improve the consistency and accuracy based on our observation. Besides, ***(D#2) Progressive Prompt*** and ***(D#3) Task Decomposition*** also help to address this challenge. Additionally, we implement ***majority voting*** by running each case multiple times and use majority voting to combat stochasticity.

We elaborate the design of (D#1 - #4) ***Post-Constraint Guided Path Analysis***, ***Progressive Prompts***, ***Task Decomposition***, and ***Self-Validation*** detailed in the rest of this section. The effectiveness and efficiency of these design strategies are rigorously evaluated in §6.2, revealing a substantial enhancement in bug detection within the Linux kernel.

4.3 D#1: Post-Constraint Guided Path Analysis

The Linux kernel frequently employs return value checks as illustrated in Table 1. Through our examination of sampled non-bug instances, we found that the sensitivity to such checks (i.e., taking them into account during the analysis) can effectively eliminate over 70% non-bug cases. In LLift, we prompt LLMs to analyze these checks, collect C_{post} and summarize the function with respect to the C_{post}. It is worth noting that current LLMs (*e.g.,* GPT-4) are not natively sensitive to the post-constraints, *i.e.,* without additional instructions, LLMs usually overlook the post-constraints. Therefore, we *teach* the LLM the rules of post-constraints through few-shot in-context learning. We elaborate the design details as follows.

4.3.1 Post-Constraints Extraction. To extract the *qualified postcondition*, we first determine the post-constraints that lead to the use of suspicious variables. We incorporate few-shot in-context learning to teach LLMs how to extract such constraints from the caller context. Table 1 demonstrates several types and examples of post-constraint in the Linux kernel. We describe how we teach them to LLMs with in-context learning.

- **Check Before Use (Type A).** The motivating example we showed is Type A; by looking at its check, the post-constraint should be $ret \geq 4$. Type A' describes a similar case while in `switch-cases`, with expected output $ret \mapsto$ `crticial_case`.

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);`

```
1  int func(int* a){
2    if(some_condi)
3      return -1;
4    *a = ... //init of a
5    return 0;
6  }
```

must_init $= \emptyset$ if:

$C_{post} = \top$ or

$\forall ps \in \{\neg\texttt{some\_condi}\} : ps \perp C_{post} \ \wedge \forall o \in \{ret \mapsto 0\} : o \perp C_{post}$

must_init $= \{a\}$ if:

$(\neg\texttt{some\_condi}) \wedge C_{post}$ or

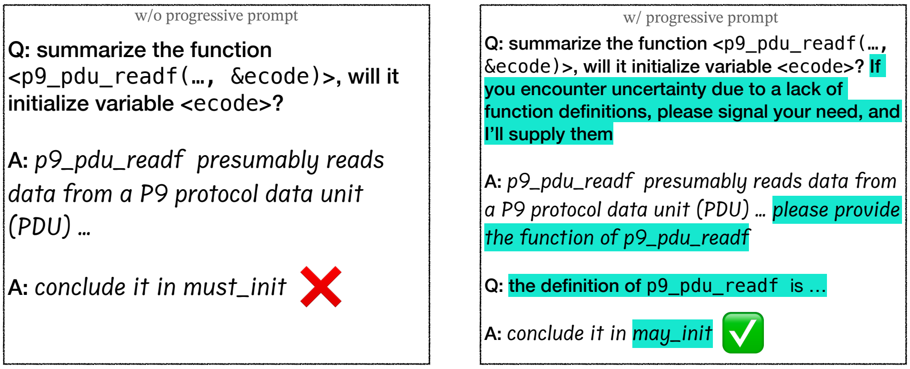
$(ret \mapsto 0) \wedge C_{post}$

Fig. 9. The func is an initializer of *a*. *a* is *may_init* or *must_init* under different post-constraints. ⊥ stands for "is disjoint from". In this function, we have the set of path constraints $\{\texttt{some\_condi}, \neg\texttt{some\_condi}\}$ and $O = \{ret \mapsto 0, ret \mapsto -1\}$, if the post-constraint is ¬come_condi (direct application) or $ret \mapsto 0$ (outcome conflict), we can prune out the path under some_condi and conclude *a* must be initialized.

- **Failure Check (Type B).** This pattern captures the opposite constraints. They are commonly used in the Linux kernel where the error conditions cause the use to become unreachable, as illustrated in Type B, the post-constraint is $err \mapsto 0$. Type B' depicts a variant where the initializer keeps retrying until success, with expected output $ret \mapsto 0$, which indicates its first successful execution to break the loop.

4.3.2 Post-Constraint Guidance. Following §3.2, Figure 9 presents a concrete example of post-constraint guided path analysis. This case shows a simple initializer $i(a)$ of the variable a. Given a potential early return at Line 3, the initialization (Line 4) may not be executed. As such, the *qualified postconditions* depend on the *post-constraints* C_{post}. Given different C_{post}, there are:

- If the use of variable a is unconditional, *i.e.*, $C_{post} = \top$, the variable a is labeled as may_init given that the initialization *may not* be reached.
  In general, if all path constraints and outcomes of must_init are *disjoint from* C_{post}, no path can be pruned out. We could also conclude a as *may_init.*
- If the use of variable a is conditional with constraints, *i.e.*, $C_{post} \neq \top$, two cases emerge:
  (1) C_{post} conflicts with the constraints of the path (*e.g.*, some_condi), or
  (2) C_{post} conflicts with the path outcome (*e.g.*, return -1).
  In these instances, C_{post} could be some_condi or func(...)==0 and we can designate *a as *must_init.*


w/o progressive prompt

Q: summarize the function <p9_pdu_readf(…, &ecode)>, will it initialize variable <ecode>?

A: *p9_pdu_readf presumably reads data from a P9 protocol data unit (PDU) …*

A: *conclude it in must_init* ❌

w/ progressive prompt

Q: summarize the function <p9_pdu_readf(…, &ecode)>, will it initialize variable <ecode>? If you encounter uncertainty due to a lack of function definitions, please signal your need, and I'll supply them

A: *p9_pdu_readf presumably reads data from a P9 protocol data unit (PDU) … please provide the function of p9_pdu_readf*

Q: the definition of p9_pdu_readf is …

A: *conclude it in may_init* ✅

Fig. 10. A demonstration of leveraging progressive prompt when interactive with ChatGPT. The conversation is simplified. Q denotes our prompt, and A denotes LLM's response. Without the progressive prompt, the LLM may miss its implementation details and conclude with an incorrect answer. We will study this case carefully in §6.5.

We provide the examples and description above to LLMs as a part of the prompt to let LLMs understand the post-constraint. This approach also utilizes few-shot in-context learning, using examples that allow LLMs to learn concepts they did not learn in training. While there are other cases of using post-constraint we have not considered, we cover most scenarios we have encountered in analyzing the Linux kernel. Besides, the few-shot in-context learning methodology is extensible, making it easy for our design to adapt to new rules and scenarios.

4.4 D#2: Progressive Prompt

The Linux kernel has a large and evolving codebase. Therefore, even though LLMs has already accumulated substantial knowledge about it during training, it may still fail to recognize some functions (e.g., newly added or modified). To make things worse, without any additional instructions, LLMs tend to guess and make up function behaviors in response to our "summarization" requirement. On the other hand, flooding the LLMs with every subroutine's source code risks exceeding their context window limits.

Inspired by recent works teaching LLMs to interact with external information and tools [Karpas et al. 2022; Parisi et al. 2022; Schick et al. 2023; Yao et al. 2023b], we apply in-context learning to teach LLMs asking for function definitions when necessary. Illustrated in Figure 10 and Figure 11, we allow the LLM to return during the analysis with *"tell me more information"*, which we then provide to enable LLM to continue the analysis.

We refer to this approach as *Progressive Prompt*; it fosters a dynamic interaction with the LLM rather than expecting a response immediately. Throughout this iterative exchange, we consistently prompt the LLM: *"If you encounter uncertainty due to a lack of function definitions, please tell your need to me, and I'll supply them"*. Should the LLM need more information, LLift will promptly extract the relevant details on demand from the source code and provide it to the LLM *automatically*.

Specifically, We teach the LLM to ask for more information with a specific format:

```
[{"type":"function_def", "name":"some_func" }]
```

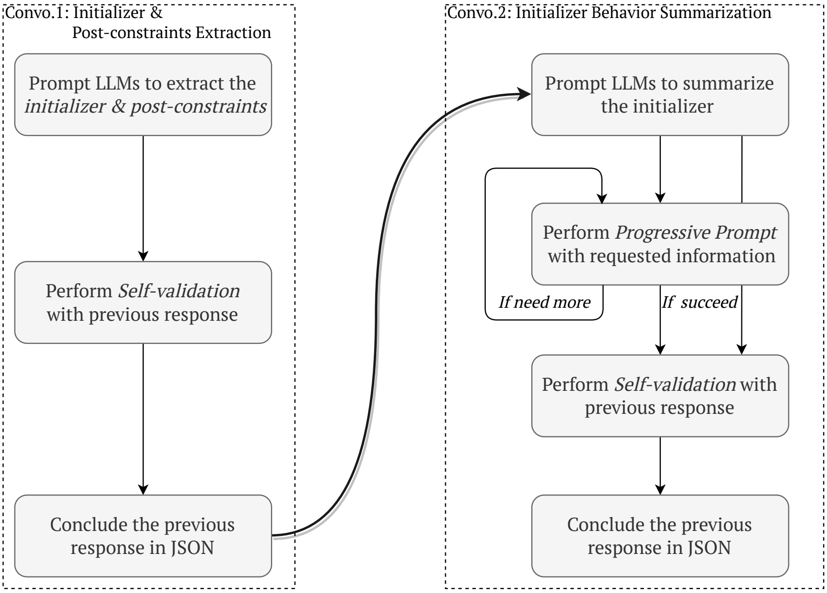

Convo.1: Initializer & Post-constraints Extraction
Prompt LLMs to extract the *initializer & post-constraints*
Perform *Self-validation* with previous response
Conclude the previous response in JSON
Convo.2: Initializer Behavior Summarization
Prompt LLMs to summarize the initializer
Perform *Progressive Prompt* with requested information
If need more
If succeed
Perform *Self-validation* with previous response
Conclude the previous response in JSON

Fig. 11. The detailed workflow of LLIFT. Each gray box represents a turn, *i.e.,* (prompt, response) pair. The dashed boxes represent a complete conversation (Convo) containing several turns.

Subsequently, LLIFT scans this format in the LLM's response. For each requested function definition, LLIFT supplies its corresponding code along with comments extracted from the Linux source code.

The iterative process continues until either the LLM no longer requests additional information or LLIFT cannot supply the requested details. In certain situations where LLIFT is unable to provide more information (*e.g.,* the definition of an indirect call), LLIFT will still prompt the LLM to proceed with the analysis. In these instances, the LLM is encouraged to infer the behavior based on the available data and its inherent knowledge, thereby facilitating continued analysis even when not all information is directly accessible.

The progressive prompt also enhances the support for unfamiliar functions. Figure10 shows its effectiveness on a real UBI bug (which UBITect missed). ChatGPT is unfamiliar with `p9_pdu_readf`, guesses its behavior from its name, and concludes an incorrect answer. Fortunately, the progressive prompt solves the case effectively. We study this case more carefully in §6.5 (Case III).

4.5 Other Prompt Design

We employ two additional prompt optimization strategies, task decomposition, and self-validation, in the design of LLIFT.

4.5.1 D#3: Task Decomposition. Task decomposition is a universal idea used in LLM prompting. As illustrated in Figure 11, we first employ a multi-conversation approach to complete the task. Each conversation essentially consists of multiple turns. Compared to combining all three subtasks into a single conversation, this division allows a more manageable and effective way of achieving the task. The efficacy of this task decomposition approach is further evaluated in §6.2.

Structured Output. Our workflow necessitates a structured output for automation. However, LLMs often produce suboptimal results when directly prompted to respond only with a structured format.

As LLMs build responses incrementally, word-by-word, based on preceding outputs [Vaswani et al. 2017], direct prompts to output JSON may interrupt their thought progression. This emphasizes the importance of first soliciting responses in natural language to ensure comprehensive and effective reasoning. Consequently, we instruct the LLM first to articulate their thought processes in English, followed by a subsequent prompt to transform their response into a JSON summary.

4.5.2 D#4: Self-Validation. LLMs can sometimes display unpredictable or inconsistent behaviors, particularly in complex scenarios involving detailed logical constructs. Consider a case where an initializer carries the postcondition `must_init` with some constraint c. LLMs may still (sometimes) mistakenly assume it to be `may_init`, despite the explicit presence of $C_{post} = c$. Also, LLMs might sometimes erroneously make up a non-existent post-constraint and incorrectly mark a `may_init` case as `must_init`.

These phenomena may stem from a lack of relevant training data, which aligns with our observation, for LLMs are often unaware of post-constraint. An Intuition is that if we prompt LLMs to review their response, they will be able to notice the previous mistakes and thus correct themselves to elicit the correct answer eventually. We refer to this practice as *self-validation.*

We also try to ***avoid false negatives*** by emphasizing *"may_init" is always a safe choice when you meet uncertain functions and code* in the self-validation. This strategy may sacrifice precision, but it tries to ensure the soundness as possible.

We implement self-validation with a series of rules, and all rules are in the form of *if statements*; for example, *"When {cases}, you should {action}"*. Self-validation does not contain any posterior knowledge. It only emphasizes some rules that should always be satisfied. Despite not bringing new information, this practice allows LLMs to correct their mistakes and yield better results. We evaluate the effect of self-validation in §6.2.

4.5.3 Additional Prompting Strategies. To further optimize the efficacy of our model, we have incorporated several additional strategies into our prompt design:

- **Chain-of-Thought.** Leveraging the Chain-of-Thought (CoT) approach, we encourage the LLMs to engage in stepwise reasoning, using the phrase *"think step by step"*. This not only helps generate longer, comprehensive responses, but it also provides intermediate results at each juncture of the thought process. Previous studies suggest the CoT approach considerably enhances the LLMs' reasoning capabilities [Chen et al. 2023a]. We incorporate the CoT strategy into every prompt.
- **Source Code Analysis.** Rather than analyzing abstract representations, we focus our attention on the functions within the source code. This approach not only economizes on token use compared to LLVM IR but also allows the model to leverage the semantic richness of variable names and other programming constructs to conduct a more nuanced analysis.

Designing effective prompts involves many intriguing nuances. For instance, the substitution of words with their synonyms can impact the final result. A striking example is the paradoxical interpretation of negations by the LLM. If prompted with a command like *"don't do something"*, the LLM occasionally comprehends it as *"do something"*, the exact opposite of the intended instruction. In our prompt design, we avoid unnecessary negations and always favor affirmative sentences.

There are still some interesting details in designing an effective prompt, but we will not list them all due to space constraints since they do not change the overall strategy. The interested reader is referred to our open-source project for further details on prompt design and implementation[1].

[1] https://sites.google.com/view/llift-open/prompt

5 IMPLEMENTATION

We implement the prototype of LLift based on OpenAI's GPT-4 API [OpenAI (2022) 2022] (gpt-4-0613). We describe some implementation details in the following aspects:

UBITect. UBITect is a prominent UBI detector for the Linux kernel. Compared to tools such as cppchecker and clang static analysis, UBITect can analyze larger scopes of code and identify more potential bugs by design [Zhai et al. 2020]. Crucially, the static analysis performed by UBITect strives for ***soundness***, leading to a large number of bug reports. Given the many undecided cases presented by UBITect, there is ample opportunity for further improvements. Hence, we target UBITect for the implementation and assessment of LLift.

Interaction with LLMs. LLift's interaction with LLMs is fully automated and managed by a simple Python script containing roughly 1,000 lines of code. In addition, it uses seven prompts, constituting about 2,000 tokens in prompts. Besides sending prompts and waiting for responses, our script also 1) interacts with LLMs in the progressive prompt design, 2) locates function definitions within the Linux source code, and 3) processes responses from LLMs, then receives and stores them in a database.

Hyper-Parameters. There are several hyper-parameters in calling the APIs provided by OpenAI. We choose `max_token` and `temperature` to 1,024 and 1.0, respectively. `max_token` controls the output length; since LLMs always predict the next words by the previous output, the longer output can benefit and allow its reasoning. However, too many tokens for single requests can quickly exhaust the context window for a long conversation (A conversation contains multiple prompts and responses pairs, so we always need to put all previous turns for next response), and thus we pick 1,024 as a balance.

The temperature controls the randomness and also the ability to reason. Intuitively, we want the analysis to be as non-random as possible and reduce the temperature (it can take a value between 0 and 2 for GPT models); however, an overly low temperature can result in repetitive or overly simplistic responses. We simply set it to 1.0 (also the default of gpt-4-0613), which allows for high-quality responses in our experiment. While carefully adjusting the temperature for each conversation or even each turn has the potential to improve results, we find that the default temperature can already produce good results.

6 EVALUATION

Our evaluation aims to address the following research questions.

- **RQ1 (Performance):** How much enhancement can LLift bring to UBITect?
- **RQ2 (Comparison):** How does the performance of individual components within LLift compare to that of the final design?
- **RQ3 (Model Versatility):** How does LLift perform when applied to LLMs other than GPT-4?
- **RQ4 (Generality):** How does LLift perform on projects other than the Linux kernel?

We evaluate these research questions on GPT-4, under API from OpenAI with version gpt4-0613. For RQ3, we also test GPT-3.5 with version gpt-3.5-turbo-0613, Bard, and Claude 2 additionally.

Evaluation target: Linux kernel. We primarily focus on the Linux kernel where UBITect was originally evaluated against. We include all potential bugs output by its static analysis stage but experienced timeout or memory exhaustion during its symbolic execution stage. Overall, UBITect's static analysis stage produced 140,000 potential bugs, with symbolic execution able to process only 60%, leaving 53,000 cases undecided, which means that these cases are generally difficult for static analysis or symbolic execution to analyze. Considering the non-existence of the ground truth of

Table 2. True positives identified by LLift from Rnd-300, with extended to 1000 cases, analyzing Linux v4.14.

Initializer	Caller	File Path	Variable	Line
read_reg	get_signal_parameters	drivers/media/dvb-frontends/stv0910.c	tmp	504
regmap_read	isc_update_profile	drivers/media/platform/atmel/atmel-isc.c	sr	664
ep0_read_setup	ep0_handle_setup	drivers/usb/mtu3/mtu3_gadget_ep0.c	setup.bRequestType	637
regmap_read	mdio_sc_cfg_reg_write	drivers/net/ethernet/hisilicon/hns_mdio.c	reg_value	169
bcm3510_do_hab_cmd	bcm3510_check_firmware_version	drivers/media/dvb-frontends/bcm3510.c	ver.demod_version	666
readCapabilityRid	airo_get_range	drivers/net/wireless/cisco/airo.c	cap_rid.softCap	6936
e1e_rphy	__e1000_resume	drivers/net/ethernet/intel/e1000e/netdev.c	phy_data	6580
pci_read_config_dword	adm8211_probe	drivers/net/wireless/admtek/adm8211.c	reg	1814
lan78xx_read_reg	lan78xx_write_raw_otp	drivers/net/usb/lan78xx.c	buf	873
t1_tpi_read	my3126_phy_reset	drivers/net/ethernet/chelsio/cxgb/my3126.c	val	193
pci_read_config_dword	quirk_intel_purley_xeon_ras_cap	arch/x86/kernel/quirks.c	capid0	562
ata_timing_compute	opti82c46x_set_piomode	drivers/ata/pata_legacy.c	&tp	564
pt_completion	pt_req_sense	drivers/block/paride/pt.c	buf	368

these 53,000 cases, We randomly chose 300 from the 53,000 cases (refers to **Rnd-300**) for evaluating LLift.

Other evaluation targets. To showcase the generality of LLift, we additionally adapted UBITect to evaluate Nginx (version 1.18.0) [Nginx 2020] and EDK II (version stable202211) [TianoCore 2022]. For Nginx, we are not aware of any known UBI bugs whereas there do exist a few in EDK II.

Turns. Due to the progressive prompt, each case may require different turns. In Rnd-300, the average number of turns is 2.78, with a max of 8 and a variance of 1.20.

Cost. On average, it costs 7,000 tokens and $0.43 in GPT-4 to analyze each potential bug in rand-300. Also, we spent about 50 human hours inspecting all results and obtain the ground truth.

6.1 RQ1: Performance

Precision Analysis. Along with the result of Rnd-300, LLift reports ten positives, and we manually determine that five are true positives. This represents a precision of 50%.

Furthermore, as shown in Table 2, we continue the running and extend to 1,000 cases. LLift reports 26 positives among the 1,000 cases overall, where 13 of them are true positives based on our manual inspection. In keeping with UBITect and focusing on the analysis of Linux v4.14 (released in 2017), three are removed, and 11 still exist in the latest Linux kernel. We confirm four of them as true bugs with the Linux community. The rest are identified as "bugs will happen only in theory" by maintainers. They follow the pattern where a UBI bug can indeed occur if some error condition occurs. However, such error conditions are related to hardware errors, which are unlikely in practice. In other words, they are still bugs but are deemed low priority.

Recall Analysis. In our detailed examination of the Rnd-300 dataset, we observe that LLift does not ***miss any real bugs***, achieving a recall rate of 100%. Despite the limited data sampled, this result indicates that integrating GPT-4 into our implementation does not introduce apparent unsoundness.

As we delve further into our research, we assess the capability of our system to detect real bugs that UBITect previously discovered using symbolic execution. Upon reviewing all 52 verified bugs highlighted by UBITect, LLift accurately *identifies every single one*. This performance shows that ***LLift empirically never misses real bugs that UBITect can discover.***

Comparing to UBITect. UBITect can either report all cases in Rnd-300 as bugs from its static analysis, causing a precision of 0.02, or ignore all of them due to the threshold of time or memory in its symbolic execution, leading to a zero recall. LLift upgrades the scope of the UBITect analysis to include those difficult cases and find more bugs in the Linux kernel. All the bugs found by

Table 3. Performance evaluation on a selected dataset (Cmp-40) of LLift with progressive addition of design components: Post-Constraint Guided Path Analysis (PGA), Progressive Prompt (PP), Self-Validation (SV), and Task Decomposition (TD). (C) indicates the number of *C*onsistent cases of each setting.

Combination	**TN(C)**	**TP(C)**	**Precision**	**Recall**	**Accuracy**	**F1 Score**
Simple Prompt	12(9)	2(1)	0.12	0.15	0.35	0.13
PGA	13(9)	5(1)	0.26	0.38	0.45	0.31
PGA+PP	5(3)	6(1)	0.21	0.46	0.28	0.29
PGA+PP+SV	5(2)	11(8)	0.33	0.85	0.40	0.48
PGA+PP+TD	22(14)	6(4)	0.55	0.46	0.70	0.50
PGA+PP+TD+SV	25(17)	13(12)	0.87	1.00	0.95	0.93
Oracle	27(27)	13(13)	-	-	-	-

UBITect have trivial post-constraints ($C_{post} = \top$) and InitStatus(v) with *may_init*. Following our methodology described in §4.3, LLift identifies all of them.

Imprecise and Failed Cases. Despite the effectiveness of LLift, there are still 13 false positives by mistakenly classifying `must_init` cases as `may_init`. Upon carefully examining these cases, we attribute the imprecision to various factors, which we discuss in detail in §6.6. Briefly, we give a breakdown of them here: *Incomplete constraint extraction* (4 cases), *Information gaps in UBITect* (5 cases), and *Missing runtime information* (4 cases). Besides, Four cases exceed the maximum context length while exploring deeper functions in the progressive prompt (*i.e.*, > 8k tokens).

6.2 RQ2: Contributions of Design Components

Dataset. In RQ2, we craft **Cmp-40** with all 13 positive cases and random 27 negative cases form 1,000 running cases (extended of Rnd-300) from RQ1.

In our effort to delineate the contributions of distinct design strategies to the final results, we undertook an evaluative exercise against the Cmp-40 dataset, employing varying configurations of our solution, each with a unique combination of our proposed strategies. As illustrated in Table 3, the strategies under consideration encompass Post-Constraint Guided Path Analysis (*PGA*), Progressive Prompt (*PP*), Self-Validation (*SV*), and Task Decomposition (*TD*). The findings underscore an overall trend of enhanced performance with the integration of additional design strategies.

In this study, the *Simple Prompt* corresponds to a straightforward prompt, *"check this code to determine if there are any UBI bugs"*, a strategy that has been found to be rather insufficient for discovering new vulnerabilities, as corroborated by past studies [Ma et al. 2023; OpenAI (2023) 2023b; Tian et al. 2023], reflecting a modest recall rate of 0.15 and a precision of 0.12.

Incorporating Post-Constraint Guided Path Analysis (*PGA*) offers a notable enhancement, enabling the LLM to uncover a wider array of vulnerabilities. As shown in Table 3, there is a substantial improvement in recall in comparison to the baseline, an anticipated outcome considering PGA's pivotal role in our solution. However, solely relying on this strategy still leaves a lot of room for optimization.

The influence of Progressive Prompt (*PP*) on the results is quite intriguing. While its impact appears to lower precision initially, the introduction of task decomposition and self-validation in conjunction with PP reveals a substantial boost in performance. Without PP, the LLM is restricted to deducing the function behavior merely based on the function context's semantics without further code analysis. Even though this approach can be effective in a range of situations, it confines the reasoning ability to the information available in its training data. By checking the detailed

Table 4. Comparison of different LLMs on real bugs, from typical of real bugs of UBITect

Caller	GPT-4	GPT-3.5	Claude2	Bard
hpet_msi_resume	✓	✓	✓	✗
ctrl_cx2341x_getv4lflags	✓	✓	✗	✗
axi_clkgen_recalc_rate	✓	✓	✓	✓
max8907_regulator_probe	✓	✓	✓	✓
ov5693_detect	✓	✓	✗	✓
iommu_unmap_page	✓	✗	✓	✗
mt9m114_detect	✓	✓	✓	✓
ec_read_u8	✓	✓	✓	✓
compress_sliced_buf	✓	✓	✗	✓

conversation, we notice the omission of TD or SV tends to result in the LLM neglecting the post-constraint, subsequently leading to errors.

Beyond influencing precision and recall, Task Decomposition (*TD*) and Self-Validation (*SV*) also play a crucial role in enhancing ***consistency***. In this context, a result is deemed *consistent* if the LLM yields the same outcome across its initial two runs. A comparison between our comprehensive final design encompassing all components, and the designs lacking TD and SV, respectively, reveals that both TD and SV notably augment the number of consistent results, and deliver 17 and 23 consistent results in its negative and positive results, respectively, underscoring their importance in ensuring reliable and consistent outcomes.

Finally, TD also holds significance in terms of conserving tokens. In our evaluation phase, we identified two instances within the PGA+PP and PGA+PP+SV configurations where the token count surpassed the limit set by GPT-4. However, this constraint was not breached in any case when TD was incorporated.

In summary, all design components of LLift, including Post-Constraint Guided Path Analysis, Progressive Prompt, Self-Validation, and Task Decomposition, contribute to its performance.

6.3 RQ3: Alternative Models

Dataset. In RQ3, we study the performance of LLift on top of alternative models with 9 typical real bugs from UBITect. As we mentioned in RQ1, these cases are more straightforward with trivial post-constraint ($C_{post} = \top$).

Table 4 provides a comprehensive view of the performance of our solution, LLift, when implemented across an array of LLMs including GPT-4, GPT-3.5, Claude 2 [Anthropic (2023) 2023], and Bard [Krawczyk and Subramanya 2023]. GPT-4 passes all tests, while GPT-3.5, Claude 2, and Bard exhibit recall rates of 89%, 67%, and 67%, respectively. Despite the unparalleled performance of GPT-4, the other LLMs still produce substantial and competitive results, thereby indicating the wide applicability of our approaches.

It is important to note that not all design strategies in our toolbox are universally applicable across all language models. Bard and GPT-3.5, in particular, exhibit limited adaptability towards the progressive prompt and task decomposition strategies. Bard's interaction patterns suggest a preference for immediate response generation, leveraging its internal knowledge base rather than requesting additional function definitions, thereby hindering the effectiveness of the progressive prompt approach. Similarly, when task decomposition is implemented, these models often misinterpret or inaccurately collect post-constraints, subsequently compromising the results. To harness their maximum potential, we apply only post-constraint guided design specifically (*i.e.*, without other design strategies) for GPT-3.5 and Bard.

In contrasting with the GPT series, Bard and Claude 2 demonstrate less familiarity with the Linux kernel and are more prone to failures due to their unawareness of the `may_init` possibility of initializers. Overall, GPT-4 remains at the pinnacle of performance for LLift, yet other LLMs can achieve promising results.

Table 5. Analysis result on Nginx. UBITect reports 11 potential bugs and all of them are false alarms. LLift analyzes all of them correctly.

Initializer	Caller	File Path	Variable	Line	Result
ngx_http_complex_value	ngx_http_split_clients_variable	http/modules/ngx_http_split_clients_module.c	val	91	✓
ngx_read_glob	ngx_conf_include	core/ngx_conf_file.c	name	866	✓
ngx_http_complex_value	ngx_http_complex_value_size	http/ngx_http_script.c	value	126	✓
ngx_strchr	ngx_http_file_cache_set_slot	http/ngx_http_file_cache.c	p+1	2432	✓
ngx_open_dir	ngx_walk_tree	core/ngx_file.c	dir	989	✓
ngx_array_init	ngx_http_compile_complex_value	http/ngx_http_script.c	sc	226	✓
ngx_parse_addr	ngx_parse_addr_port	core/ngx_inet.c	text	671	✓
ngx_file_size	ngx_conf_parse	core/ngx_conf_file.c	&cf->conf_file->file	589	✓
ngx_cpymem	ngx_conf_parse	core/ngx_conf_file.c	dump->last	609	✓
ngx_pstrdup	ngx_conf_include	core/ngx_conf_file.c	file	873	✓
ngx_calloc_buf	ngx_http_range_body_filter	http/modules/ngx_http_range_filter_module.c	b->pos	932	✓

Table 6. Analysis results on EDK-2 (CryptoPkg). The first five cases are real bugs identified by LLift, the rest are false alarms of UBITect, LLift successfully recognizes 12 out of 13 among them.

Initializer	Caller	File Path	Variable	Line	Result
ASN1_get_object	Asn1GetTag	.../Pk/CryptX509.c	ObjTag	1903	✓
ASN1_get_object	Asn1GetTag	.../Pk/CryptX509.c	ObjCls	1904	✓
ASN1_get_object	X509GetTBSCert	.../Pk/CryptX509.c	Asn1Tag	842	✓
ASN1_get_object	X509GetCertFromCertChain	.../Pk/CryptX509.c	Asn1Tag	1838	✓
ASN1_get_object	X509VerifyCertChain	.../Pk/CryptX509.c	Asn1Tag	1741	✓
gRT->GetTime	time	.../SysCall/TimerWrapper.c	Time.Year	97	✓
sscanf	ipv4_from_asc	.../crypto/x509v3/v3_utl.c	a,b,c,d	1091	✓
ssl_get_security_level_bits	ssl_security_default_callback	.../ssl/ssl_cert.c	level	908	✓
strtoul	do_tcreate	.../crypto/asn1/asn_mstbl.c	*eptr	77	✓
X509V3_get_value_bool	alg_module_init	.../crypto/evp/evp_cnf.c	m	39	✓
ASN1_get_object	d2i_ASN1_OBJECT	.../crypto/asn1/a_object.c	tag	226	✓
strtoul	bitstr_cb	.../crypto/asn1/asn1_gen.c	*eptr	752	✓
ASN1_INTEGER_get_int64	ASN1_INTEGER_get	.../crypto/asn1/a_int.c	r	547	✓
x509_object_idx_cnt	X509_STORE_CTX_get1_crls	.../crypto/x509/x509_lu.c	cnt	618	✓
OBJ_find_sigid_algs	x509_sig_info_init	.../crypto/x509/x509_set.c	mdnid	209	✓
PEM_get_EVP_CIPHER_INFO	pem_bytes_read_bio_flags	.../crypto/pem/pem_lib.c	cipher	255	✓
sl2	FE	.../crypto/aria/aria.c	y.c	1082	✗
skip_prefix	equal_nocase	.../crypto/x509v3/v3_utl.c	*pattern	611	✓

6.4 RQ4: Results on Projects Other Than Linux

Albeit the implementation and experiment of LLift primarily focuses on the Linux kernel, our workflow should also work in other C projects. But since the Linux kernel is a well-known open-source project with significant documentation that has been part of GPT's training dataset, we are curious whether our LLM-based solution can work well for other projects.

As mentioned, we chose two other projects: Nginx (version 1.18.0) [Nginx 2020] and EDK II (version stable202211) [TianoCore 2022].

Table 5 shows the results on Nginx, UBITect reports 11 bugs, and all of them are not real bugs by our manual check. LLift also identifies all of them as *must_init*, consistent with our manual check.

Table 6 shows the results for EDK II. Since UBITect produces a large number of bug reports, we focus on a sub-module (CryptoPkg) where there are known UBI bugs. LLift performs well on this project, identifies all actual bugs, and reports only one false positive (*i.e.,* it treats a *must_init* case as *may_init*) with regard to the initializer `sl2`. We find that the false positive is due to the lack of struct definitions. Specifically, the variable `y.c` is an array and `sl2` initializes the first four elements. However, without knowing the exact definition of the struct, LLift can only give a conservative result of *may_init*. Adapting the progressive prompt to fetch more information, *e.g.,* structure definitions, could resolve this issue.

Notably, these real bugs in EDK II are aligned with a recent patch[2] (ObjTag and Asn1Tag) on March 2023, which was not learned by gpt-4-0613 in its training. These Nginx and EDK II results show the generality of LLift that can effectively work even in less popular projects.

6.5 Case Study

In this case study, we pick three interesting cases demonstrating the effectiveness of LLift in analyzing function behaviors and detecting uninitialized variables. These cases contain difficult patterns for program analysis, loops, and callbacks with function pointers. Symbolic execution of UBITect is not capable of analyzing these examples; however, with the help of LLMs, LLift can complete the analysis successfully. We also put the complete conversations on the anonymous online page for reference.

Loop and Index. Figure 12 presents an intriguing case involving the variable `pages[j]`, which is reported by UBITect as used in Line 13 potentially without being initialized. Unfortunately, this case is a false positive, which is hard to prune due to loops. Specifically, the initializer function `get_user_pages_unlocked()`, which is responsible for mapping user space pages into the kernel space, initializes the `pages` array allocated in Line 3. If `get_user_pages_unlocked()` is successfully executed, `pages[0]` through `pages[res-1]` pointers will be initialized.

To summarize the behavior, *i.e.,* `must_init` facts under conditions where the use is reachable, we must first extract the *post-constraints* that lead to the use of `pages`. Through interacting with ChatGPT, LLift successfully extracts it:

```
{
    "initializer": "res =
  ↪  get_user_pages_unlocked(uaddr, nr_pages,
  ↪  pages, rw == READ ? FOLL_WRITE : 0)",
    "suspicious": ["pages[j]"],
    "postconstraint": "res < nr_pages && res > 0
  ↪  && j < res",
}
```

After feeding the post-constraints to LLM, LLift then successfully obtains the result:

```
{
    "ret": "success",
    "response": {
        "must_init": ["pages[j]"],
        "may_init": [],
    }
}
```

```
1  static int sgl_map_user_pages(...){
2  ...
3  if ((pages = kmalloc(...)) == NULL)
4   return -ENOMEM;
5  res = get_user_pages_unlocked(..., pages);
6   /* Errors and no page mapped ... */
7  if (res < nr_pages)
8    goto out_unmap;
9  ...
10 out_unmap:
11  if (res > 0) {
12   for (j=0; j < res; j++)
13    put_page( pages[j] );
14    res = 0;
15  }
16  kfree(pages);
17 }
```

Fig. 12. Case Study I (Loop and Index). Derived from `drivers/scsi/st.c`

[2] https://patchew.org/EDK2/20230324162146.588-1-mikuback@linux.microsoft.com/20230324162146.588-5-mikuback@linux.microsoft.com

```
1  static int hv_pci_enter_d0(struct hv_device
   ↪  *hdev){
2    ...
3    init_completion(&comp_pkt.host_event);
4    pkt->completion_func =
     ↪  hv_pci_generic_compl;
5    pkt->compl_ctxt = &comp_pkt;
6    ...
7
8    wait_for_completion(&comp_pkt.host_event);
9
10   if ( comp_pkt.completion_status  < 0)
11   ...
12 }
13 static void hv_pci_generic_compl(void
   ↪  *context, ...){
14   struct hv_pci_compl *comp_pkt = context;
15
16   if (resp_packet_size >= offsetofend(...))
17     comp_pkt->completion_status =
       ↪  resp->status;
18   else
19     comp_pkt->completion_status = -1;
20
21   complete(&comp_pkt->host_event);
22 }
```

Fig. 13. Case Study II (Concurrency and Indirect Call). Derived from drivers/pci/host/pci-hyperv.c

```
int p9_check_zc_errors(...){
  ...
  err = p9pdu_readf(req->rc,
  ↪  c->proto_version, "d", &ecode);
  err =  -ecode ;
  ...
}
int p9pdu_readf(struct p9_fcall *pdu, int
↪  proto_version, const char *fmt, ...)
  ...
  ret = p9pdu_vreadf(pdu, proto_version,
  ↪  fmt, ap);
  ...
  return ret;
}
int p9pdu_vreadf(..., va_list ap){
   switch (*fmt) {
     case 'd':{
       int32_t *val = va_arg(ap, int32_t *);
       if (pdu_read(...)) {
               errcode = -EFAULT;
               break;
       }
       val = ...;  // initialization
   }
   return errcode;
}
```

Fig. 14. Case Study III (Unfamiliar Function), derived from net/9p

As we can see, GPT-4 exhibits impressive comprehension of this complex function. It perceives the variable pages[j] being used in a loop that iterates from 0 to res-1. This insight leads GPT-4 to correctly deduce that all elements in the pages array must be initialized, *i.e.*, they are must_init. This example underscores GPT-4's proficiency in handling loop and even index sensitivity.

Concurrency and Callback. Consider the case illustrated in Figure 13. At first glance, UBITect flags Line 10 for potentially using the variable comp_pkt.completion_status before initialization.

The function's body seemingly lacks any code that initializes it, leading UBITect to report it as a potential bug. However, the mystery unravels when we examine hv_pci_generic_compl(), the actual initializer function assigned to pkt in Line 4. The variable in question is indeed initialized, but intriguingly, its initializer emerges from a concurrent function instead of within its own thread. Here wait_for_completion() is a synchronization primitive that pauses the current thread and waits for the new thread (*i.e.*, hv_pci_generic_compl()) to complete. Despite this complexity, GPT-4 adeptly navigates the concurrency and callback handling, pinpointing the accurate initializer and outputting a precise result.

It is worth noting that we do not encode any knowledge about the Linux kernel synchronization primitives. LLift prompts LLMs with *"The 'initializer' must be the 'actual' function that initializes the variable."* and then LLMs can automatically identify the function `hv_pci_generic_compl()` as the initializer of `comp_pkt.completion_status`.

Unfamiliar Function. As outlined in Section §2.3, Large Language Models (LLMs) inherently recognize the semantics of widely-used functions, such as `sscanf()`. Some critics contend that the ubiquity of online documents explains why LLMs are familiar with them, suggesting the models absorb this knowledge during training and act as a search engine [Chiang 2023]. However, our study shows that LLMs perform well even with unfamiliar functions.

Figure 14 presents an intriguing real-world bug. The function `p9pdu_readf()`, used in a specific submodule in the Linux kernel is much rarer than `sscanf()`.

Compared to our motivating example, it lacks a check of its return value, leaving the parameter `ecode` at risk of being uninitialized, *i.e.,* if `pdu_read()` returns non-zero in line 19 (thus "break" early). As mentioned earlier in §4.4, this case prompts GPT-4 requesting related function definitions and then generates a summary. Furthermore, LLift also pinpoints that `ecode` could be initialized when `p9pdu_readf()` returns 0, demonstrating the efficacy of LLift for unfamiliar cases. The result is as follows:

```
{
  "initializer":  "err = p9pdu_readf(req->rc, c->proto_version, 'd', &ecode)",
  "suspicious": ["ecode"],
  "postconstraint": null,
  "response": {
    "must_init": [],
    "may_init": [{ "name": "ecode", "condition": "p9pdu_readf returns 0"}]
  }
}
```

6.6 Reason for Imprecision

Despite LLift achieving a precision about 50% in real-world applications, the precision can still be improved in the future. Some can be solved with better prompts or better integration with static analysis.

Information Gaps in UBITect. LLift is deliberately decoupled from UBITect in its design, which creates some information gaps. For instance, UBITect does not provide explicit field names within a structure when a specific field is in use. This information gap can result in LLift lacking precision in its analysis. This challenge can be addressed with focused engineering efforts to enrich the output information from UBITect.

Variables With Same Name. In general, the LLM usually confuses different variables in different scopes (*e.g.*, different function calls) with same name. For example, if the suspicious variable is `ret` and passed as an argument to its initializer (say, `func(&ret)`) and there is another stack variable defined in `func` also called `ret`, LLMs can confuse them. Explicitly prompting and teaching LLM to note the difference does not appear to work. One solution is to leverage a simple static analysis to normalize the source code to ensure each variable has a unique name.

Indirect Call. As mentioned §4.4, LLift follows a simple but imprecise strategy to handle indirect calls. Theoretically, existing static analysis tools, such as MLTA [Lu and Hu 2019], can give possible targets for indirect calls. However, each indirect call may have multiple possible targets and dramatically increase the token usage. We leave the exploration of such an exhaustive strategy for future work. LLift may benefit from a more precise indirect call resolution.

Additional Constraints. There are many variables whose values are determined outside of the function we analyze, *e.g.,* preconditions capturing constraints from the outer caller. Since our analysis is fundamentally under-constrained, this can lead LLift to incorrectly determine a `must_init` case to be `may_init`. Mitigating this imprecision relies on analysis with a larger scope to provide more information.

7 LIMITATIONS

Dependency on Closed-Source LLMs. Our work's reproducibility is potentially limited due to its reliance on GPT-4, a closed-source API with frequent updates. We specified the latest version during the paper writing, gpt-4-0613. However, at the moment of the paper's publication, the GPT had multiple updates, and the model we used in this paper might be discarded as early as June 2024. This dependency could potentially threaten external validity. Nevertheless, our experiment showed that while not as perfect as GPT-4, other LLMs such as Bard and Claude 2, can still produce meaningful results. In the future, we will also experiment on open-sourced LLMs that can be trained and fine-tuned locally, such as Llama 2 [Touvron et al. 2023], to facilitate reproducibility.

Data Bias from UBITect. The potential data bias introduced by UBITect could threaten the internal validity of our study. Although the design of LLift is decoupled with UBITect, we only make tests on top of it and its design and implementation may introduce bias into our experiment. LLift may perform differently on other static analysis tools.

8 DISCUSSION AND FUTURE WORK

Post-Constraint Guidance with LLMs. The integration of post-constraint guided path analysis and the analysis with LLMs in our system is a calculated design choice, underpinned by their individual strengths and collective synergy. At its core, the post-constraint-guided approach is an effective tool for optimizing and narrowing the scope of path exploration. While it excels in this domain, comprehending and interpreting complex code semantics are still challenging.

Conversely, LLMs have consistently demonstrated prowess in code comprehension. Their ability to reason about intricate code constructs makes them invaluable in scenarios that demand nuanced understanding. By migrating this reasoning capability of LLMs to a static analysis framework, we aim to bridge the gap between broad path optimization and in-depth code understanding.

Better Integration with Static Analysis. Our proposed solution operates independently of the static analysis methods, taking only the bug report from static analysis. Looking into the future, we can consider integrating static analysis and LLMs in a holistic workflow. For example, this could involve selectively utilizing LLMs as an assistant to overcome certain hurdles encountered by static analysis, *e.g.,* difficulty in scaling up the analysis or summarizing loop invariants. In turn, further static analysis based on these findings can provide insights to refine the queries to the LLM. This iterative process could enable a more thorough and accurate analysis of complex cases. We believe such a more integrated approach is a promising direction.

From UBI to Other Bugs. Our decision to commence with Use-Before-Initialization (UBI) stemmed from its inherent simplicity. Nevertheless, the foundational techniques we have crafted, particularly encapsulating a function's data flow with post-constraint guidance, indicate promising adaptability for a spectrum of other bug types, such as use-after-free, out-of-bound read/write, and taint flow analysis. While the transition from UBI to these more nuanced vulnerabilities will undoubtedly introduce complexities, we envision a future ripe with opportunities for broadening the scope and efficacy of our methodology.

9 RELATED WORK

Techniques of Utilizing LLMs. Wang et al. [2023] propose an embodied lifelong learning agent based on LLMs. Pallagani et al. [2023] explores the capabilities of LLMs for automated planning. Weng [2023] summarizes recent work in building an autonomous agent based on LLMs and proposes two important components for planning: *Task Decomposition* and *Self-reflection*, which are similar to the design of LLift. Beyond dividing tasks into small pieces, task decomposition techniques also include some universal strategies such as Chain-of-thought Wei et al. [2023] and Tree-of-thought Yao et al. [2023a]. The general strategy of self-reflection has been used in several flavors: ReAct Yao et al. [2023b], Reflexion Shinn et al. [2023], and Chain of Hindsight Liu et al. [2023]. Despite the similarity in name, self-reflection is fundamentally *different* from self-validation in LLift where the former focuses on using external sources to provide feedback to their models. Huang et al. [2022] let an LLM self-improve its reasoning without supervised data by asking the LLM to lay out different possible results. These techniques inspire us to leverage LLMs to build LLift.

LLMs for Vulnerability Detection. Sun et al. [2024] proposes GPTScan, combining GPT with static analysis to detect logic vulnerabilities in smart contracts. While both GPTScan and LLift utilize LLMs and static analysis, our approach addresses the challenges of static analysis in uncovering vulnerabilities in large codebases (e.g., Linux kernel) and employs LLMs to augment its effectiveness. In contrast, GPTScan uses static analysis to confirm vulnerabilities reported by LLMs in smart contracts, which have only few hundreds of lines of code typically. Khare et al. [2023] uses LLMs to perform dataflow analysis and detect vulnerabilities, showcasing superior performance compared to traditional static analysis. In contrast, LLift explores how to synergize the strengths of both LLMs and static analysis for vulnerability detection. To the best of our knowledge, LLift is the first to demonstrate how to apply LLMs to overcome the limitations of static analysis and enhance its bug-finding capabilities.

LLMs for Program Analysis. Ma et al. [2023] and Sun et al. [2023] explore the capabilities of LLMs when performing various program analysis tasks such as control flow graph construction, call graph analysis, and code summarization. They conclude that while LLMs can comprehend basic code syntax, they are somewhat limited in performing more sophisticated analyses such as pointer analysis and code behavior summarization. In contrast to their findings, our research with LLift has yielded encouraging results. We conjecture that this might be due to several reasons: (1) benchmark selection, *i.e.,* Linux kernel vs. others. (2) Prompt designs. (3) GPT-3.5 vs. GPT-4.0 – prior work only evaluated the results using only GPT-3.5. Pei et al. [2023a] prompt LLMs to focus on reasoning about loop invariants with decent performance. In contrast, LLift leverages LLMs for program behavior summarization and integrates LLMs successfully into a static analysis pipeline.

LLMs for Software Engineering. Xia and Zhang [2023] propose an automated conversation-driven program repair tool using ChatGPT, achieving nearly 50% success rate. Ahmed et al. [2024] show that code comprehension tasks can benefit from the semantics produced by static analysis on the code. Pearce et al. [2023] examine zero-shot vulnerability repair using LLMs and find promise in synthetic and hand-crafted scenarios but face challenges in real-world examples. Chen et al. [2023b] teach LLMs to debug their own predicted programs to increase the correctness but only perform on relatively simple programs. Pei et al. [2023b] proposes a new architecture of LLMs for program reasoning tasks. Lemieux et al. [2023] leverages LLM to generate tests for uncovered functions when the search-based approach got coverage stalled. Feng and Chen [2023] use LLM to replay Android bugs automatedly. LangChain [2023] propose LangSimith, an LLM-powered platform for debugging, testing, and evaluating. These diverse applications underline the vast potential of LLMs in software engineering. LLift complements these efforts by demonstrating the efficacy of LLMs in bug finding in the real world.

10 CONCLUSION

In our study, we present LLift, a pioneering framework that augments static analysis with LLMs for effective and efficient UBI bug detection. Using *post-constraint guided analysis*, LLift enhances path verification capabilities, addressing complex vulnerabilities. By leveraging a set of strategies in prompting, LLift engages LLMs effectively, ensuring reliable and consistent outputs. Our tests reveal LLift's proficiency, identifying 13 new UBI bugs in the Linux kernel with a 50% precision. This research underscores the promising fusion of LLMs with static analysis and lays a foundation for future exploration in this area.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the National Science Foundation under Grant No. 1953933 and 1652954. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

DATA-AVAILABILITY STATEMENT

We put all the code (for §4-5) and data for the experiment (§6) on Zenodo [Li et al. 2024]: https://doi.org/10.5281/zenodo.10780591 and GitHub: https://github.com/seclab-ucr/LLift. Besides, we also provide an online page to show our prompts and case studies: https://sites.google.com/view/llift-open

REFERENCES

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *2024 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Anthropic (2023). 2023. Claude 2. https://www.anthropic.com/index/claude-2

Jiuhai Chen, Lichang Chen, Heng Huang, and Tianyi Zhou. 2023a. When do you need Chain-of-Thought Prompting for ChatGPT? http://arxiv.org/abs/2304.03262 arXiv:2304.03262 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. Teaching Large Language Models to Self-Debug. http://arxiv.org/abs/2304.05128

Ted Chiang. 2023. ChatGPT Is a Blurry JPEG of the Web. *The New Yorker* (Feb. 2023). https://www.newyorker.com/tech/annals-of-technology/chatgpt-is-a-blurry-jpeg-of-the-web Section: annals of artificial intelligence.

Copilot. 2023. GitHub Copilot documentation. https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-for-individuals

Sidong Feng and Chunyang Chen. 2023. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. https://doi.org/10.48550/arXiv.2306.01987 arXiv:2306.01987 [cs].

Anjana Gosain and Ganga Sharma. 2015. Static Analysis: A Survey of Techniques and Tools. In *Intelligent Computing and Applications (Advances in Intelligent Systems and Computing)*, Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi (Eds.). Springer India, New Delhi, 581–591.

J. Huang. 2007. *Path-Oriented Program Analysis.* https://doi.org/10.1017/CBO9780511546990

Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. 2022. Large Language Models Can Self-Improve. http://arxiv.org/abs/2210.11610 arXiv:2210.11610 [cs].

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *Comput. Surveys* 55, 12 (Dec. 2023), 1–38. https://doi.org/10.1145/3571730

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. 2022. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. https://doi.org/10.48550/arXiv.2205.00445 arXiv:2205.00445 [cs]

Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. http://arxiv.org/abs/2311.16169 arXiv:2311.16169 [cs].

Jack Krawczyk and Amarnag Subramanya. 2023. Bard's latest update: more features, languages and countries. https://blog.google/products/bard/google-bard-new-features-update-july-2023/

LangChain. 2023. Announcing LangSmith, a unified platform for debugging, testing, evaluating, and monitoring your LLM applications. https://blog.langchain.dev/announcing-langsmith/

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. (2023).

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. *Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach (Artifact)*. https://doi.org/10.5281/zenodo.10780591

Hao Liu, Carmelo Sferrazza, and Pieter Abbeel. 2023. Chain of Hindsight Aligns Language Models with Feedback. http://arxiv.org/abs/2302.02676 arXiv:2302.02676 [cs].

Kangjie Lu and Hong Hu. 2019. Where Does It Go?: Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London United Kingdom. https://doi.org/10.1145/3319535.3354244

Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. http://arxiv.org/abs/2305.12138 arXiv:2305.12138 [cs].

Nginx. 2020. nginx. https://nginx.org/en/

OpenAI (2022). 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt

OpenAI (2023). 2023a. Function calling and other API updates. https://openai.com/blog/function-calling-and-other-api-updates

OpenAI (2023). 2023b. GPT-4 Technical Report. http://arxiv.org/abs/2303.08774 arXiv:2303.08774 [cs].

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. http://arxiv.org/abs/2203.02155 arXiv:2203.02155 [cs].

Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Biplav Srivastava, Lior Horesh, Francesco Fabiano, and Andrea Loreggia. 2023. Understanding the Capabilities of Large Language Models for Automated Planning. http://arxiv.org/abs/2305.16151 arXiv:2305.16151 [cs].

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. TALM: Tool Augmented Language Models. https://doi.org/10.48550/arXiv.2205.12255 arXiv:2205.12255 [cs]

Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2022. A Survey of Parametric Static Analysis. *ACM Comput. Surv.* 54, 7 (2022), 149:1–149:37. https://doi.org/10.1145/3464457

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP46215.2023.00001

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023a. Can Large Language Models Reason about Program Invariants?. In *Proceedings of the 40th International Conference on Machine Learning*.

Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. 2023b. Symmetry-Preserving Program Representations for Learning Code Semantics. http://arxiv.org/abs/2308.03312 arXiv:2308.03312 [cs].

Luke Salamone. 2021. What is Temperature in NLP? https://lukesalamone.github.io/posts/what-is-temperature/ Section: posts.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. https://doi.org/10.48550/arXiv.2302.04761 arXiv:2302.04761 [cs]

Jessica Shieh. 2023. Best practices for prompt engineering with OpenAI API | OpenAI Help Center. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-openai-api

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. http://arxiv.org/abs/2303.11366 arXiv:2303.11366 [cs].

Yisheng Song, Ting Wang, Puyu Cai, Subrota K. Mondal, and Jyoti Prakash Sahoo. 2023. A Comprehensive Survey of Few-shot Learning: Evolution, Applications, Challenges, and Opportunities. *Comput. Surveys* 55, 13s (July 2023), 271:1–271:40. https://doi.org/10.1145/3582688

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic Code Summarization via ChatGPT: How Far Are We? http://arxiv.org/abs/2305.12865 arXiv:2305.12865 [cs].

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *2024 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant – How far is it? http://arxiv.org/abs/2304.11938 arXiv:2304.11938 [cs].

TianoCore. 2022. tianocore/edk2. https://github.com/tianocore/edk2

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. http://arxiv.org/abs/2307.09288 arXiv:2307.09288 [cs].

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. http://arxiv.org/abs/2305.16291 arXiv:2305.16291 [cs].

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. http://arxiv.org/abs/2201.11903 arXiv:2201.11903 [cs].

Lilian Weng. 2023. LLM-powered Autonomous Agents. *lilianweng.github.io* (Jun 2023). https://lilianweng.github.io/posts/2023-06-23-agent

Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. http://arxiv.org/abs/2304.00385

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, San Diego CA USA, 1–10. https://doi.org/10.1145/3520312.3534862

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. http://arxiv.org/abs/2305.10601 arXiv:2305.10601 [cs].

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. *International Conference on Learning Representations (ICLR)* (2023).

Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: A Precise and Scalable Method to Detect Use-before-Initialization Bugs in Linux Kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL]

Shen Zheng, Jie Huang, and Kevin Chen-Chuan Chang. 2023. Why Does ChatGPT Fall Short in Providing Truthful Answers? http://arxiv.org/abs/2304.10513 arXiv:2304.10513 [cs].

Received 21-OCT-2023; accepted 2024-02-24