Poster: Assisting Static Analysis with Large Language Models: A ChatGPT Experiment

Haonan Li
UC Riverside
hli333@ucr.edu

Yu Hao
UC Riverside
yhao016@ucr.edu

Yizhuo Zhai
UC Riverside
yzhai003@ucr.edu

Zhiyun Qian
UC Riverside
zhiyunq@cs.ucr.edu

***Abstract*—Recent advances of *Large Language Model*s (LLMs), e.g., ChatGPT, exhibited strong capabilities of comprehending and responding to questions across a variety of domains. Surprisingly, ChatGPT, a general LLM, even possesses a strong understanding of program code. In this paper, we investigate where and how LLMs can assist static analysis by asking appropriate questions. In particular, we target a specific bug-finding tool, which produces many false positives from the static analysis. Interestingly, in our evaluation, we find that these false positives can be effectively pruned by asking carefully constructed questions about function-level behaviors or function summaries. Specifically, with a case study of 16 false positives due to imprecise function summaries, we find that ChatGPT (GPT-3.5) can successfully prune 8 out of 16, whereas GPT-4 can successfully 16 out of 16. We find LLMs a promising tool that can enable a more effective and efficient program analysis.**

***Index Terms*—Static analysis, Bug finding, Large language model, ChatGPT**

I. Introduction

Static analysis plays a crucial role in uncovering vulnerabilities and enhancing software quality. Unfortunately, static analysis faces an inherent tradeoff between precision and scalability. In practice, static analysis tools often produce a large number of false positives, hampering their adoption.

In this paper, we investigate the feasibility of taking advantage of *Large Language Model*s (LLMs), such as ChatGPT, as flexible and general assistance to prune such false positives. In particular, we hypothesize that ChatGPT can generate function summaries that are more precise than what static analysis can compute, e.g., in the presence of loops and operations on variable-length data structures (e.g., `strlen()`). These precise function summaries are the basis for pruning false positives.

We develop an automated, interactive process for leveraging ChatGPT to generate accurate, precise, structured function summaries. We evaluate our approach on 16 false positive cases due to imprecise function summaries reported from UBITect [1] using two versions of ChatGPT (GPT-3.5 and GPT-4). Our results show the resulting summaries from GPT-3.5 are not always correct and precise, which leads to half of the false positives being pruned. Surprisingly, GPT-4 attains both precise and correct function summaries for all cases, which can prune 100% of the false positives. A side benefit of using ChatGPT is that function summaries are generated within a predictable amount of time (unlike static analysis).

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

Figure 1: Code snippet of `sscanf` and its usecase, derived from Linux kernel

This research highlights the potential of LLMs in enhancing the effectiveness and efficiency of static analysis.

II. Background

UBITect targets *Use Before Initialization* (UBI) bugs in the Linux kernel through a two-stage process [1]. The first stage employs a bottom-up summary-based static analysis of the kernel. Essentially, the analysis is a *MAY* analysis, where function summaries indicate potential bug occurrences, resulting in a large number of warnings (*i.e.,* ∼140k) In the second stage, UBITect uses symbolic execution to filter out false positives by verifying the path feasibility of reported bugs. However, over 40% of the reported warnings are discarded due to timeout or memory limitations in symbolic execution, potentially rejecting genuine bugs.

III. Motivation and Overview

A. Motivating Example

Figure 1 shows a false positive produced by UBITect. A bug is reported in line 4 and line 5 because it is believed that arguments `a`, `b`, `c`, `d` are not initialized but used. However, both are incorrect due to the following reasons:

- **Inability to recognize special functions.** First, the report in line 4 is incorrect because there is in fact no "use" of `args` inside `sscanf()`, other than the `va_start()` call and `va_end()` call in line 9 and line 11. Unfortunately, UBITect cannot find the definition of these two functions and conservatively assumed that they might "use" `args`. However, these functions are compiler's built-in ones that simply recognize variable-length arguments and no "use" is involved. Indeed, the semantic of `sscanf()` is to "define"/write new values into `args` as opposed to "use".

- **Unawareness of post-conditions**. Second, the report in line 5 is incorrect because the function summary generated by UBITect is insensitive to the return value check, or post-conditions. Specifically, UBITect does not know the arguments `a`, `b`, `c`, `d` are always initialized if the return value is greater than or equal to 4. Instead, it simply conservatively estimates that all function parameters "may" be left uninitialized, demonstrating the lack of sensitivity and flexibility when function summaries are computed.

B. Observations

We argue that both challenges in the static analysis are prevalent. The variable-length argument issue can be attributed to *Inherent Knowledge Boundaries* (KB), where static analysis often needs to encode domain knowledge of various kinds, e.g., modeling certain special functions that involve assembly code or recursive data structures. The unawareness of post-conditions can be attributed to *Tradeoff Between Precision and Scalability* (TPS). Unfortunately, computing function summaries precisely with respect to different sensitivity requirements, especially in the presence of loops, variable-length data structures, etc., is simply infeasible. This is why we often see functions such as `strlen()` being modeled and summarized manually.

We believe the advent of LLMs [2] offers a promising alternative for function summarization. LLMs can efficiently generate knowledgeable and precise function summaries by leveraging extensive training data. They encompass a vast knowledge base and recognize complex patterns, enabling the production of more precise and comprehensive summaries.

IV. Methodology

Our objective is to employ LLMs to generate precise function summaries. The generated summaries should be structured and easily integrated into existing systems.

Specifically, our approach asks ChatGPT to generate function summaries about what arguments are initialized in a given function call, considering relevant contexts such as concrete arguments that are passed to the function call and return value checks. We engage in an interactive process, allowing ChatGPT to request additional information when needed. Finally, we prompt ChatGPT to produce a structured summary for easy integration.

ChatGPT is known to generate unreliable answers with low confidence. To mitigate this issue, we design an automatic interaction mechanism to avoid forcing ChatGPT into giving uncertain answers. Specifically, we prompt ChatGPT initially with, "*if you feel uncertain due to a lack of code definition, you should respond with what additional function/structure definitions you need*". In response, ChatGPT can reply in the following format: `["type": "function_def", "name": "some_func"]`. We then utilize a script to automatically locate the definition of the function `some_func`, provide it to ChatGPT, and prompt it to reanalyze the case.

Currently, the GPT-4 API is not available (as of April 2023), so our script is based on GPT-3.5, with GPT-4 results being generated manually. However, there are no inherent challenges in supporting GPT-4 once the API becomes available.

Table I: Selected function summaries: "S?" for Soundness and "C?" for Completeness. Type indicates analysis challenges: *Inherent Knowledge Boundaries* (KB), *Tradeoff Between Precision and Scalability* (TPS), or both.

Function Call	Type	GPT-3.5		GPT-4	
		S?	C?	S?	C?
sscanf	KB, TPS	✓	✓	✓	✓
read_mii_word	KB, TPS	✗	✗	✓	✓
acpi_decode_pld_buffer	KB, TPS	✓	✓	✓	✓
of_graph_get_remote_node	KB	✓	✓	✓	✓
msr_read	KB	✓	✓	✓	✓
cpuid	KB	✗	✗	✓	✓
bq2415x_i2c_read	KB	✓	✓	✓	✓
parse_nl_config	TPS	✓	✗	✓	✓
snd_interval_refine	TPS	✗	✗	✓	✓
xfs_iext_lookup_extent	TPS	✓	✗	✓	✓
__skb_header_pointer	TPS	✗	✗	✓	✓
snd_rawmidi_new	TPS	✓	✗	✓	✓
snd_hwdep_new	TPS	✗	✗	✓	✓
xdr_stream_decode_opaque_inline	TPS	✓	✓	✓	✓
of_parse_phandle_with_args	TPS	✓	✓	✓	✓
kstrtoul	TPS	✓	✓	✓	✓

```
"func_call": "sscanf(str, \"%u.%u.%u.%u%n\",
                &a, &b, &c, &d, &n) >= 4",
"must_init": ["&a", "&b", "&c", "&d"],
```

Figure 2: Snippet of the summary of `sscanf(...)>=4`.

V. Evaluation

To evaluate our approach, we randomly select 19 false alarms from UBITect. We determine that 16 of these cases are due to imprecise summaries (*i.e.,* KB or TPS), while the other three cases stem from indirect calls or heap variables, which are distinct from imprecise function summaries. Among the 16 cases, ChatGPT can directly (*i.e.,* without requiring additional information such as function definitions) summarize three of them (`sscanf`, `cpuid`, `kstrtoul`).

In assessing the outcomes of function summaries, our attention is centered on two primary aspects: ***Soundness***, i.e., whether variables identified as "must_init" are correct; and ***Completeness***, i.e., whether all "must_init" variables are correctly identified. Each case is executed three times, and if any of the runs exhibit unsound or incomplete, we mark it with a ✗. Table I presents a comparison between the function summaries generated by GPT-3.5 and GPT-4. As we can see, GPT-3.5's results show that only 68.75% are sound, and 50% are complete. Conversely, GPT-4 exhibits significantly enhanced performance. Our evaluation underscores the practical utility of our approach for generating precise function summaries in program analysis.

References

[1] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. Yu, "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*, ser. ESEC/FSE 2020, 2020.

[2] OpenAI (2023), "GPT-4 Technical Report," Mar. 2023, arXiv:2303.08774 [cs]. [Online]. Available: http://arxiv.org/abs/2303.08774