We find LLMs a promising tool that can enable a more effective and efficient program analysis. 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. 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 pilot study of 20 false positives, we can successfully prune 8 out of 20 based on GPT-3.5, whereas GPT-4 had a near-perfect result of 16 out of 20, where the four failed ones are not currently considered/supported by our questions, e.g., involving concurrency. Additionally, it also identified one false negative case (a missed bug). We find LLMs a promising tool that can enable a more effective and efficient program analysis.

1 INTRODUCTION

Static analysis faces the inherent trade-off between precision and scalability [13]. In real-world applications, static analysis tools often generate a significant number of false positives, hindering their widespread adoption [3, 7, 22, 23].

This paper explores the possibility of employing Large Language Models (LLMs), such as ChatGPT [15], as versatile and comprehensive aids to static analysis. Specifically, ChatGPT even shows a capability in understanding programming language [4] and we conjecture that it can generate function summaries with greater precision than those computed by static analysis, particularly in the presence of loops and operations on variable-length data structures (e.g., strlen()). These precise function summaries serve as the foundation for more effective analysis that reduces both false positives and false negatives.

We develop a systematic methodology that utilizes ChatGPT to create accurate summaries of functions automatically. Our approach has been evaluated on false positives and false negatives, identified as imprecise function summaries by a real-world static analysis tool known as UBITect [21]. Notably, using the latest GPT-4 model, our method has provided exact summaries for 16 of 20 instances in our pilot study, effectively eliminating false positive cases.

We summarize our contributions as follows:

- We develop a novel approach utilizing LLM to enhance function summary precision and reduce both false positives and false negatives in static analysis.
- We propose an automated and progressive methodology for generating precise function summaries with ChatGPT.
- We evaluate our approach to complement a real-world static analysis tool, which showed great promise.
- To foster further research and development, we open source our work on https://github.com/seclab-ucr/GPT-Expr.
static int libcfs_ip_str2addr(...);
if (sscanf(str, "%u.%u.%u.%un", &a, &b, &c, &d) >= 4 && ...
    va_start(args, fmt);
    i = sscanf(buf, fmt, args);
    va_end(args);
}

Figure 1: Code snippet of sscanf and its use case, derived from Linux kernel

UBITect. UBITect targets Use Before Initialization (UBI) bugs in the Linux kernel through a two-stage process [21]. The first stage employs a bottom-up summary-based static analysis of the kernel. The analysis is a MAY analysis, where function summaries indicate potential bug occurrences, resulting in many bugs (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 bugs are discarded due to timeout or memory limitations in symbolic execution, potentially rejecting genuine bugs. In this paper, we focus on these 40% discarded cases to prune out false positives and also find missed actual bugs.

3 MOTIVATION

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 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 the compiler’s built-in ones that 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 postconditions. Second, the report in line 5 is incorrect because the function summary generated by UBITect is insensitive to the check of its return value (if(sscanf(...))==4), or post-condition [11]. Therefore, UBITect provides a conservative summary and estimates all parameters “may” left uninitialized.

3.1 Observation

In light of our motivating sscanf case, we argue that both issues are prevalent in static analysis. The variable-length argument issue can be attributed to Inherent Knowledge Boundaries (KB), and the unawareness of post-conditions is essentially due to the Exhaustive Path Exploration (PE) in path-sensitive static analysis.

Exhaustive Path Exploration. Correctly handling cases like sscanf() requires it to consider the check: sscanf(...)>=4. Unfortunately, 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 a combinatorial explosion. The vast array of execution paths necessitates the exploration of myriad functions, many of which ultimately prove irrelevant to the specific analysis task at hand. In the sscanf() case, its return value is computed inside an unbounded loop when iterating over an unknown string variable buf. This causes UBITect’s symbolic execution to time out exactly due to this problem.

The advent of LLMs [1] offers a promising alternative to bypass these challenges. This is because LLMs, especially ChatGPT being trained and aligned with extensive materials that include both natural language and program codes and shows a promising understanding of code comprehension [12].
We then ask ChatGPT whether the variable "must" be initialized, given the calling context and under specific post-conditions. Lastly, if needs more info, we make the question smaller.

Prompt Design. Prompting ChatGPT to elicit reliable responses is essential [16]. Based on our experience, we identify the following key principles for designing prompts when summarizing function behaviors related to variable initialization.

- **Chain-of-Thought.** The Chain-of-Thought (CoT) [5] approach utilizes the phrase "think step by step" to encourage ChatGPT to generate longer responses containing intermediate results at each step. We incorporate the CoT strategy into our prompts.

- **Task Decomposition.** Referring to §4.1, we find that mixing explicit and implicit information is equivalent to "not a bug" and we make the question smaller.

- **Progressive Prompt.** In instances where our requirement states, "Always deliver a result", ChatGPT may produce unreliable responses. To circumvent this issue, we develop the progressive prompt. As Figure 3 demonstrates, we progressively provide simpler the question and the smaller the scope, the more likely ChatGPT will be able to provide a correct answer.

**5 EVALUATION**

To evaluate the effectiveness of our approach, we randomly sample a number of inconclusive cases from the symbolic execution phase of UBITect, as shown in Figure 2. Specifically, we randomly select 20 cases that were manually determined to be false positives and two additional real bugs missed by UBITect. Because all of these cases are inconclusive using UBITect alone, we are interested in assessing the effectiveness of our approach in determining the outcomes of these reported bugs. All experiments (both for GPT-3.5 and GPT-4) are run under ChatGPT version on March 23, 2023.

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. We perform three runs for each case to account for the probabilistic nature of ChatGPT’s output, and if any of the runs exhibit unsound or incomplete, we consider the result of the case to be failed.

**5.1 Naive Approach**

To address the research question, we gather three real UBI CVEs and input them into ChatGPT (GPT-4) for analysis. As mentioned earlier in §4.1, the results of providing the context directly are poor. In our experiments, we explicitly mention the uninitialized variable and ask ChatGPT to determine the existence of a genuine bug. The three CVEs examined are CVE-2022-1016, CVE-2022-0382, and CVE-2021-29647, with the first two disclosed after the cut-off date (September 2021).

We also apply prompt design strategies mentioned in §4.2. For example, we leverage the progressive prompt by requesting ChatGPT with "If you need function definitions, you should ask us, and we will provide them." We test the naive approach on 3 real CVEs, repeating each test case 5 times. Our findings reveal that none of the test cases were consistently analyzed correctly across all five repetitions. For the three CVEs, ChatGPT correctly identifies the bug in 3/5, 2/5, and 0/5 instances, respectively.

**5.2 Results**

Table 1 compares the function summaries generated by GPT-3.5 and GPT-4. Most responses are consistent across all three runs. We excluded four false positive cases from the table because UBITect reported them for reasons beyond what we outlined in §3.1, e.g., inaccurate indirect call resolution. As we can see, GPT-3.5’s results show that only 61% are sound, and 44% are complete. On the other hand, GPT-4 achieves 77% soundness and 57% completeness, highlighting the improvement in the quality of responses.

---

1https://github.com/seclab-ucr/GPT-Expr/blob/main/conversation.md
Table 1: Selected function summaries: “S?” for Soundness and “C?” for Completeness. Type indicates analysis challenges: Inherent Knowledge Boundaries (KB), Exhaustive Path Exploration (PE), or both.

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Type</th>
<th>GPT-3.5</th>
<th>GPT-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sscanf</td>
<td>KB, PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>read_mii_word</td>
<td>KB, PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>ACPI decode pld buffer</td>
<td>KB, PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>of_graph_get_remote_node</td>
<td>KB</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>msr_read</td>
<td>KB</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>cpuid</td>
<td>KB</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>bq2415x_l2c_read</td>
<td>KB</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>parse_nl_config</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>snd_interval_refine</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>xsf_iext_lookup_extent</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>__skb_header_pointer</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>snd_rawmidi_new</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>snd_hwdep_new</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>xdr_stream_decode</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>of_parse_phandle...</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>kstrtoul</td>
<td>PE</td>
<td>✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗</td>
</tr>
</tbody>
</table>

False Positives of UBITect

Example 2: Figure 5 shows another UBI bug in net/9p/client.c. We can see that function p9pdu_vreadf() may not initialize its parameter ecode when it returns -EFAULT. Nonetheless, p9_check_zc_errors() directly uses its value without checking the return value at Line 3. While ChatGPT always correctly identifies the relevant code (Line 8-14 in Figure 5), its final verdict is occasionally incorrect (one in three times) — categorizing ecode as “must_init”. This inconsistency between the obtained results and the reasoning steps is a known issue in chain-of-thought prompting [19, 24]. In future work, we plan to employ additional design strategies to address this inconsistency [9, 10, 18].

5.3 Case Study

In this case study, we discuss two examples (both are real UBI bugs) demonstrating the effectiveness and limitations of our approach in analyzing function behaviors and detecting uninitialized variables.

Example 1: Figure 4 presents a real bug in arch/x86/kvm/lapic.c, where the uninitialized variable val is used. If pv_eoi_get_user returns a value less than 0, the code continues without an early return, leading to Line 5, which is used as val1&0x1. UBITect fails to detect this bug due to timeout. ChatGPT handles the case and correctly identifies the bug by categorizing val as “may_init”.

Example 2: Figure 5 shows another UBI bug in net/9p/client.c. We can see that function p9pdu_vreadf() may not initialize its parameter ecode when it returns -EFAULT. Nonetheless, p9_check_zc_errors() directly uses its value without checking

6 DISCUSSION & LIMITATIONS

We recognize several limitations in our current implementation. Our experiments have been conducted on a relatively small scale, primarily due to the unavailability of the GPT-4 API, which necessitates manual testing. Nevertheless, our workflow is fully automatic by design and can work in large-scale datasets directly with the API. Furthermore, our approach does not yet consider indirect calls or more complicated types of bugs, we left them in the future. We have not encountered token limit issues in our experiment. This might imply the current context window (i.e., 8k tokens for GPT-4) is sufficient for most cases. However, given the progressive prompt design, we suspect they may reach the limitation when ChatGPT continuously requesting for more functions.

Recent announcements suggest that Bard can understand code effectively [2]. However, our preliminary tests indicate that it performs worse than ChatGPT. Specifically, it consistently provides results directly rather than progressively requesting unknown function definitions.

7 CONCLUSION

In this work, we present a novel approach that utilizes ChatGPT to aid in static analysis, which has yielded promising results. We believe our effort only scratched the surface of the vast design space, and hope our work will inspire future research in this exciting space.
REFERENCES


Received 2023-05-03; accepted 2023-07-19