Assisting Static Analysis with Large Language Models: A ChatGPT Experiment

Haonan Li
hli333@ucr.edu
UC Riverside
Riverside, California, USA

Yu Hao
yhao016@ucr.edu
UC Riverside
Riverside, California, USA

Yizhuo Zhai
yzhai003@ucr.edu
UC Riverside
Riverside, California, USA

Zhiyun Qian
zhiyunq@cs.ucr.edu
UC Riverside
Riverside, California, USA

ABSTRACT

Recent advances of *Large Language Models* (LLMs), *e.g.,* ChatGPT, exhibited strong capabilities of comprehending and responding to questions across a variety of domains. Surprisingly, ChatGPT 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. 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.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

static analysis, bug detection, large language model

ACM Reference Format:
Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613078

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613078

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.

2 BACKGROUND & RELATED WORK

LLM for Software Engineering. Xia *et al.* [20] propose an automated conversation-driven program repair tool using ChatGPT, achieving nearly 50% success rate. Pearce *et al.* [14] examine zero-shot vulnerability repair using LLMs and found promise in synthetic and hand-crafted scenarios. Lemieux *et al.* [8] leverages LLM to generate tests for uncovered code. In this paper, we explore how LLM can be used as an alternative to achieve better results when static analysis encounters difficulties.

```
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 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.

Inherent Knowledge Boundaries. Static analysis often needs to encode domain knowledge to model certain special functions which cannot be analyzed. Beyond the variable-length argument case, there are numerous other scenarios (especially in the Linux kernel) that involve complex domain knowledge and are difficult to analyze directly, such as assembly code, hardware behaviors, concurrency, and compiler built-in functions [6].

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].

4 METHODOLOGY

Our aim is to integrate ChatGPT with UBITect to enhance the detection of UBI bugs. The process is depicted in Figure 2. It's important to remember that UBITect operates in two phases: an initial static analysis to pinpoint potential UBI bugs, followed by symbolic execution to confirm these suspicions. However, in our replicated experiments, 40% of cases were disregarded due to either time or memory restrictions. This scenario presents a conundrum: if we classify these potential bugs as false positives, we run the risk of overlooking real bugs. Conversely, if we consider them as true bugs, we may face an overwhelming influx of false positives. To navigate this challenge, our approach is to consult ChatGPT on these ambiguous cases, enabling us to discern if they are false positives or genuine bugs.

At a high level, we first extract key facts of a potential UBI bug from UBITect, *i.e.,* the uninitialized variable, the post-condition, and the function that might initialize the variable. Next, we automatically put this information into ChatGPT and let it determine whether the variable is being initialized.

4.1 Design Exploration

Before achieving the final design, we first explore a straightforward method by asking ChatGPT at one time: given the code context, we ask ChatGPT whether a potential UBI bug is an actual bug. Specifically, we provide a detailed description of what constitutes a UBI bug, and then copy the function where the uninitialized variable is declared. We also follow the principles outlined in §4.2 such as Chain-of-Thought.

We test this idea against two case studies along with three real CVEs that we refer to in the later evaluation. The two cases include a false positive (`cpuid`), and the other is a true bug (`p9pdu_readf`). ChatGPT failed in both cases. Interestingly, even though we explicitly offered ChatGPT to ask follow-up questions about function definitions, we found that it would still generate a plausible but incorrect answer without requesting more information.

This has motivated us to consider breaking down the task into smaller and simpler ones for ChatGPT to solve. Intuitively, the

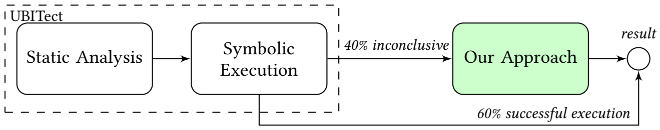

Figure 2: The overview of our approach. Start with the discarded cases by UBITect and determine whether these reported bugs are true or false.

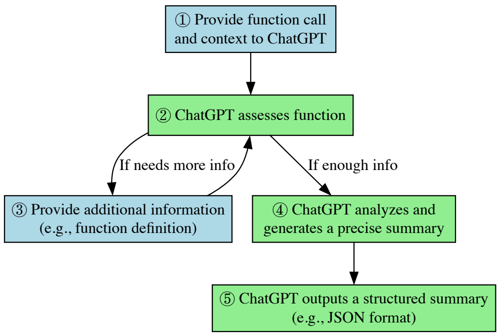

Figure 3: The workflow. Green stands for what ChatGPT (API) responds (role: assistant) and the blue stands for what the user (script, role: user) prompts.

simpler the question and the smaller the scope, the more likely ChatGPT will be able to provide a correct answer.

4.2 Our Approach

Recognizing the crucial role function summaries play in UBITect and their direct impact on the results (as shown in the motivating example); instead of directly asking for the existence of UBI bug, we prompt ChatGPT to summarize the function. Hence, the "must_init" is equivalent to "not a bug" and we make the question smaller.

As Figure 3 shows, for each reported bug, we extract the function call context, including concrete arguments and return value checks. We then ask ChatGPT whether the variable "must" be initialized given the calling context and under specific post-conditions. Lastly, we prompt ChatGPT to generate a structured summary for seamless integration with further analysis.

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 multiple tasks is ineffective. Therefore, we break down our problem into multiple steps and instruct ChatGPT to complete smaller ones. Similarly, when we need a structured output, we always initiate a new request at the end of the conversation and prompt ChatGPT to conclude with JSON format separately.
- **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 information, such as function definitions when necessary. Specifically, we always prompt ChatGPT with the following message: "*If you experience uncertainty due to insufficient function definitions, please indicate the required functions*". Upon receiving a request for additional information from ChatGPT, we automatically retrieve and provide the required data from the source code of Linux, enabling the model to reevaluate and generate an improved response.

Limited by space, we showcase an automatic and complete interaction with ChatGPT on a webpage[1].

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 [17] 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

[1] https://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.

Function Call	Type	GPT-3.5		GPT-4	
		S?	C?	S?	C?
False Positives of UBITect					
`sscanf`	KB, PE	✓	✓	✓	✓
`read_mii_word`	KB, PE	✗	✗	✓	✓
`acpi_decode_pld_buffer`	KB, PE	✓	✓	✓	✓
`of_graph_get_remote_node`	KB	✓	✓	✓	✓
`msr_read`	KB	✓	✓	✓	✓
`cpuid`	KB	✗	✗	✓	✓
`bq2415x_i2c_read`	KB	✓	✓	✓	✓
`parse_nl_config`	PE	✓	✗	✓	✓
`snd_interval_refine`	PE	✗	✗	✓	✓
`xfs_iext_lookup_extent`	PE	✓	✗	✓	✓
`__skb_header_pointer`	PE	✗	✗	✓	✓
`snd_rawmidi_new`	PE	✓	✗	✓	✓
`snd_hwdep_new`	PE	✗	✗	✓	✓
`xdr_stream_decode...`	PE	✓	✓	✓	✓
`of_parse_phandle...`	PE	✓	✓	✓	✓
`kstrtoul`	PE	✓	✓	✓	✓
False Negatives of UBITect					
`pv_eoi_get_user`	PE	✗	✗	✓	✓
`p9pdu_readf`	KB, PE	✗	✗	✗	✗

hand, GPT-4 demonstrates a substantial enhancement in performance, attaining 94% soundness and completeness. The table's top 16 cases indicate false alarms, while the final two cases represent authentic UBI bugs.

Various reasons contribute to the inability of GPT-3.5 to analyze some cases. For example, variable name matters. In the `snd_interval_refine` case, GPT-3.5 is confused about the names of formal and actual parameters, resulting in an incorrect response.

In addition, in the GPT-3.5 experiment, we find that it would often prematurely conclude an answer without requesting more information (*e.g.,* asking for more function definitions), even though we explicitly instruct it to do so. When this happens, results are often unreliable. In contrast, GPT-4 consistently engages in lengthy conversations as needed, contributing to its superior performance.

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 `val&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

```
1 static bool pv_eoi_get_pending(struct kvm_vcpu *vcpu){
2   u8 val;
3   if (pv_eoi_get_user(vcpu, &val) < 0)
4     apic_debug(...);
5   return val & 0x1;
6 }
```

```
1 "response": {
2   "func_call": "pv_eoi_get_user(vcpu, &val) < 0",
3   "parameters": ["vcpu", "&val"],
4   "must_init": [],
5   "may_init": [{"name": "&val", "condition": ...}]
6 }
```

Figure 4: The code and summary of `pv_eoi_get_user` from GPT-4

```
1  int p9_check_zc_errors(...){
2    err = p9pdu_readf(..., "d", &ecode);
3    err = -ecode;
4  }
5
6  int p9pdu_vreadf(..., const char *fmt, va_list ap){
7     switch (*fmt) {
8       case 'd':{
9         int32_t *val = va_arg(ap, int32_t *);
10        if (pdu_read(...)) {
11                errcode = -EFAULT;
12                break;
13        }
14        val = ...;  // initialization
15    }
16    return errcode;
17 }
```

Figure 5: Code snippet of `p9pdu_readf` and its usecase, derived from `net/9p`

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].

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

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

[2] Paige Bailey. 2023. Code and debug with Bard. https://blog.google/technology/ai/code-with-bard/

[3] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* (2010), 66–75.

[4] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. http://arxiv.org/abs/2303.12712 arXiv:2303.12712 [cs].

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

[6] Clang. 2023. *Clang Language Extensions*. https://clang.llvm.org/docs/LanguageExtensions.html#builtin-functions

[7] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681.

[8] 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).

[9] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. arXiv:2308.00245 [cs.SE]

[10] Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful Chain-of-Thought Reasoning. arXiv:2301.13379 [cs.CL]

[11] Bertrand Meyer. 1997. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall. http://www.eiffel.com/doc/oosc/page.html

[12] 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].

[13] 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

[14] 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

[15] John Schulman, Barret Zoph, Jacob Hilton Christina Kim, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, Rapha Gontijo Lopes, Shengjia Zhao, Arun Vijayvergiya, Eric Sigler, Adam Perelman, Chelsea Voss, Mike Heaton, Joel Parish, Dave Cummings, Rajeev Nayak, Valerie Balcom, David Schnurr, Tomer Kaftan, Chris Hallacy, Nicholas Turley, Noah Deutsch, Vik Goel, Jonathan Ward, Aris Konstantinidis, Wojciech Zaremba, Long Ouyang, Leonard Bogdonoff, Joshua Gross, David Medina, Sarah Yoo, Teddy Lee, Ryan Lowe, Dan Mossing, Joost Huizinga, Roger Jiang, Carroll Wainwright, Diogo Almeida, Steph Lin, Marvin Zhang, Kai Xiao, Katarina Slama, Steven Bills, Alex Gray, Jan Leike, Jakub Pachocki, Phil Tillet, Shantanu Jain, Greg Brockman, and Nick Ryder. 2022. ChatGPT: Optimizing Language Models for Dialogue. (2022). https://openai.com/blog/chatgpt/.

[16] 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

[17] 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.

[18] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*.

[19] 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].

[20] 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

[21] 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)*.

[22] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V. Krishnamurthy, Trent Jaeger, and Paul L. Yu. 2022. Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-249/

[23] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically Discovering High-Order Taint Style Vulnerabilities in OS Kernels. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 811–824. https://doi.org/10.1145/3460120.3484798

[24] 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]

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