CG-Bench: Can Language Models Assist Call Graph Construction in the Real World?

Ting Yuan[*]
Huawei Technologies Co., Ltd
Beijing, China
yuanting1994@gmail.com

Wenrui Zhang
Huawei Technologies Co., Ltd
Hangzhou, China
zhangwenrui@u.nus.edu

Dong Chen[†]
Huawei Technologies Co., Ltd
Beijing, China
jameschennerd@gmail.com

Jie Wang
Huawei Technologies Co., Ltd
Beijing, China
jiewmurphy@163.com

Abstract

Language models for coding are shifting their focus from function-level to repository-level, with complex function invocations. We introduce CG-Bench , the first manually constructed benchmark that measures the ability to understand call graphs for language models. This benchmark contains 104 call sites and related code snippets associated with call chains from 7 representative open-source C/C++ projects. Language models are tasked with inference the calling targets from them. We evaluated four popular language models on CG-Bench . Surprisingly, all four models with different prompt settings achieve accuracy greater than 50% and Deepseek-6.7b with few-shot prompts reaches 69.70%. We further show four findings from a micro study, which demonstrates that using language models for call graph construction is promising and the performance can be improved by prompt hacking, removing irrelevant information, etc.

CCS Concepts: **• Computing methodologies → Artificial intelligence**; **• Software and its engineering → Software organization and properties**.

Keywords: Program analysis, Language models, Call graph, Benchmark

ACM Reference Format:
Ting Yuan, Wenrui Zhang, Dong Chen, and Jie Wang. 2025. CG-Bench: Can Language Models Assist Call Graph Construction in the Real World?. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3759425.3763379

[*]The research and writing of this work were carried out by the author during his employment at Huawei.
[†]Corresponding author

1 Introduction

Language Models (LMs) show promising capabilities for accomplishing coding tasks. Copilot can help generate 46% of the code for their users by line suggestions [24]. For function-level synthesis, the state-of-the-art Pass@1 reaches 94.4% on the HumanEval dataset [34]. However, code generation by LMs can rarely go beyond a single function. Whether LMs can handle coarser-grained coding tasks with complex caller-callee relations remains unclear.

Recently, researchers proposed a number of benchmarks for coarser-grained coding tasks. CoderEval extends function-level generation by inserting project-level context into the prompt [32]. ClassEval focuses on single class generation with average lines of code 45.7 [11]. LMs accept the class skeleton and generate its missing parts, mainly the bodies of methods in a class. SWE-bench focuses on evaluating LMs' ability to resolve real-world GitHub issues. LMs are tasked to generate patches from the issue description and codes from repo [16]. From their result, the generated patch modifies only two functions and rarely goes beyond a single file on average.

All three benchmarks consider the complex relations between functions in different forms: project-level context, class context, or under a specific issue. We still cannot tell directly from their measurements how well LMs understand the basic caller-callee relations.

To bridge this gap, we propose CG-Bench [1], which evaluates the ability of LMs to resolve the targets of a call site. We spend approximately 200 hours composing 104 cases from 7 popular C/C++ open-source projects. For each case, we provide related code snippets and the expected call targets to serve as the ground truth. With CG-Bench , we evaluated four popular open source code LMs and the resolving rates of all four LMs exceed 50%. Furthermore, we conduct a

[1]https://github.com/timmyyuan/CG-Bench

micro case study and find that language models could better utilize certain information that is missed by traditional program analysis tools. But on the contrary, with irrelevant information in prompt or bad prompt template, the inference performance will drop.

2 CG-bench

Rice's theorem states that all non-trivial semantic properties of programs are undecidable [21]. Unfortunately the call graph construction problem belongs to this kind of non-trivial problem and there is no perfect solution. Thus we rely on human experts to manually extract the chains from all selected code repositories.

2.1 Benchmark construction

Repository selection. We selected 7 open-source C/C++ projects that are frequently adopted in previous researches [5, 6, 12]. [2] Table 1 shows their project description and basic statistics. These projects are from various domains in order to cover diverse program patterns. They cover multiprocessing, complex data structures, macros, dynamic linking, etc. The sizes of their codebases range from 120 KLOC to 6,200 KLOC [10]. They contain 18,007 indirect function calls in total and 11,226 calls are from *gcc*. From these indirect call sites, we select cases for CG-Bench .

Call site sampling. As Table 1 shows, the number of indirect call sites is not propositional to the size of the code base. We use random sampling and human search to make our selection cover diverse patterns and, at the same time, be representative.

(1) *Random sampling with static analysis:* Blindly random sampling with a low sampling rate may likely hurt the diversity of patterns that appear in a call chain. To solve this problem, we rely on two static analysis tools with different analysis approaches. One is SVF [26] which uses pointer analysis and has higher precision. The other is MLTA [18] which utilizes type matching and has higher recall. For each call site, we first run both tools and compare their outputs. Then, we divide the call sites into three sets. Set A (Equals) contains the call sites that both tools generate the same result. Set B (Intersect) contains the call sites that have at least one common result from the results from SVF and MLTA. Set C (Disjoint) contains the call sites that SVF and MLTA produce totally different results. We randomly choose five call sites from each set for each repo and try to determine the correct callees for them manually. In this step, we get a total of 105 candidate cases.

(2) *Search with human experience:* However, with the help of the static analysis tools, we realize that the sampled call sites still uncover some patterns. Thus, we invite five persons with experience in program analysis and LMs to search for call sites that cover diverse patterns from the seven projects. For each project, we sampled 10 candidate cases. Therefore, a total of 70 candidate call sites are selected.

Call site filtering. As no tool can precisely drive the targets of the sampled indirect call sites, we rely on human experts to filter the samples with the following strategies: (1) The case is skipped if a human expert can't figure out any correct callee in 20 minutes. In our experience, the most time-consuming part is tracking data flows involved complex data structures (e.g. maps and sets). (2) Invocations of constructors and destructors are skipped because they may not be visible at the source level. We ignore the call relationship involving configurations and environment variables since finding such callees requires a deep understanding of the software itself. (3) We omit test code and assembly code, which is consistent with the common strategies of typical call graph construction tools.

Finally, we get 61 cases in random sampling, while in human experience searching, we obtained 43 cases. The number of selected cases for each project is shown in column #Cases in Table 1.

Related snippets extraction. In the ideal case, we should feed the entire repo and the select call site to query LMs about its targets. The current context length of LMs usually ranges from 4k to 32k [1, 13, 28, 29]. Though Claude supports up to 200k context window and Gemini supports 1 million tokens, direct repo feed is still beyond the RoPE [2, 27]. Due to context length limitation, we only reserve related code snippets for each indirect call site based on the following rules: (1) We manually trace the data flow backward from the indirect call site. Data flows originating from the same file are regarded as a code snippet. All related functions will be added to the collection of snippets. If a function body contains more than 50 lines of code, we will only keep the code that contributes to resolving the final target functions. Otherwise, we keep all code for functions. (2) We include related global definitions and macros. (3) For related code in conditional compilation directive `#ifdef`, we only reverse one implementation. Generally, different directives not change the complexity of call chains, but usually introduce more similar code. (4) For invisible code, such as calls to library functions or calls dependent on complex frameworks, we trace along the data flow to the definition of the library function or related structure, without further deep tracking.

2.2 Task formulation

Model Input. A model is given an indirect call site and its associated snippets. The model then identifies all the candidate functions that the function pointer could call. Three types of prompts are adopted in our CG-Bench , which are zero-shot prompt, improved zero-shot prompt (more clear and more formatted), and few-shot prompt. Our designed prompt templates are listed in Appendix A.

[2]We start from C/C++ projects and will include other projects form other languages in the future.

Table 1. Selected projects and statistics for the extracted cases in CG-Bench

Repository	Description	# Lines (KLOC)	# Indirect calls	# Cases	# Code snippets	Length of snippets (min/max/average)
openssh [20]	Tool for remote login with the SSH protocol	120	128	13	53	4/ 43/16.6
curl [25]	Tool for transferring data with URLs	182	1,193	17	61	1/ 52/14.9
redis [23]	In-memory data structure store	200	460	19	78	1/ 30/12.7
zfs [3]	File system with volume management capabilities	380	566	18	97	1/278/21.5
wrk [14]	HTTP benchmarking tool	601	1,241	14	77	2/113/19.5
ffmpeg [19]	Tool to record, convert and stream audio and video	1,257	3,153	17	55	2/158/19.3
gcc [8]	Well-known compiler	6,200	11,266	6	30	2/ 98/20.3
Total		8,940	18,007	104	451	N/A

Evaluation Metric. The task of constructing call graphs can be regarded as a multi-label classification task, where each pointer corresponds to a list of ground-truth labels (names of functions called by the pointer). Four common metrics (as the first row of columns 3 to 6 of Table 2 shown) for classification tasks are adopted to evaluate the performance of LMs on this task. In our task, accuracy is a binary score with 1 indicating the generated function names exactly match the manually analyzed result. Precision represents the proportion of matched function names out of all generated functions, while recall represents the ratio of correctly generated function names to the total manually analyzed functions. For each metric, we calculate the average across the single sample metrics to obtain the final score.

Evaluation Setup. We run our experiment on two Nvidia V100 GPUs with four open-sourced code LMs, including CodeLlama2-7B-Instruct, CodeLlama2-13B-Instruct [22], Deepseek-Coder-6.7b-Instruct and Deepseek-Coder-7B-Instruct-v1.5 [15]. For the sake of brevity and clarity in subsequent discussions throughout this paper, these models are respectively denoted by the abbreviated forms: CodeLlama-7b, CodeLlama-13b, Deepseek-6.7b, and Deepseek-7b-v1.5.

3 Initial Results and Findings

3.1 Overall results

Table 2 shows the summarized evaluation results for four different LMs with three different prompt templates. All LMs achieved accuracy above 50% with the zero-shot prompt. By improving the zero-shot prompt template, the accuracy improvements range from 2.02% to 11.11%. CodeLlama-13b performs the best with an accuracy of 67.68%, recall at 81.65%, precision at 83.41%, and an F1-score at 80.38%. With few-shot prompts, the performance improvements of Deepseek-6.7b are larger than CodeLlama-13b. Deepseek-6.7b achieves the highest ranking with subtle performance benefits.

From Table 2, we can see that different prompts can lead to significant variations in the performance of LMs. After rephrasing the zero-shot prompt (A.1) to a more clear and standard zero-shot prompt (A.2), all evaluation metrics have been significantly improved. After adding some examples, all models show improvement across all metrics as well, with accuracy values improving to between 59.60% and 69.70%. Overall, Table 2 indicates that it is promising to assist call graph construction with LMs with proper prompt engineering.

Table 2. Performance of 4 open-sourced LMs on CG-Bench with 3 types of prompts. Zero-shot* represents the improved zero-shot prompt.

Prompt	Model	Accuracy	Recall	Precision	F1-Score
Zero-shot	CodeLlama-7b	57.58%	75.79%	75.37%	73.11%
	CodeLlama-13b	61.62%	73.37%	79.65%	75.13%
	Deepseek-6.7b	52.53%	63.16%	71.04%	65.46%
	Deepseek-7b-v1.5	51.52%	62.79%	71.04%	65.22%
Zero-shot*	CodeLlama-7b	59.60%	77.63%	75.91%	74.29%
	CodeLlama-13b	67.68%	81.65%	83.41%	80.38%
	Deepseek-6.7b	63.64%	75.39%	81.53%	76.61%
	Deepseek-7b-v1.5	58.59%	72.20%	72.34%	70.94%
Few-shot	CodeLlama-7b	67.68%	82.06%	85.19%	81.87%
	CodeLlama-13b	68.69%	81.83%	83.81%	81.45%
	Deepseek-6.7b	69.70%	82.63%	87.85%	83.58%
	Deepseek-7b-v1.5	59.60%	69.22%	74.59%	70.66%

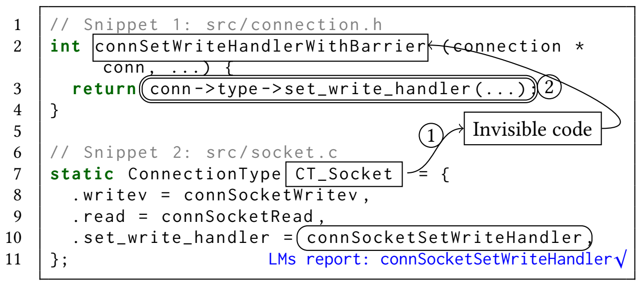
3.2 Case Study

To understand the capability of language models (LMs) for call graph construction, we analyze representative cases in CG-Bench in detail. Figures 1 and 2 depict six cases [3]. We focus on the patterns in call chains where LMs excel or face challenges.

To present these cases more clearly, we use ▢ to enclose the target function, ▣ to signify the invocation of indirect functions, and ▭ to represent the variables that propagate function pointers. The arrows between them illustrate the direction of data flow, with circled numbers indicating the flow steps. With these cases, we have the following findings:

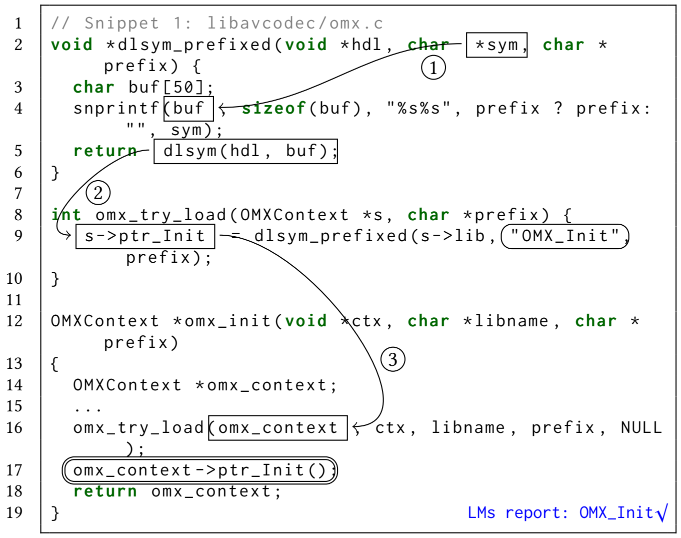
Finding 1: *LMs can utilize the information that is ignored or hard to be leveraged by traditional static analysis tools.*

[3]For additional examples, please refer to Appendix B



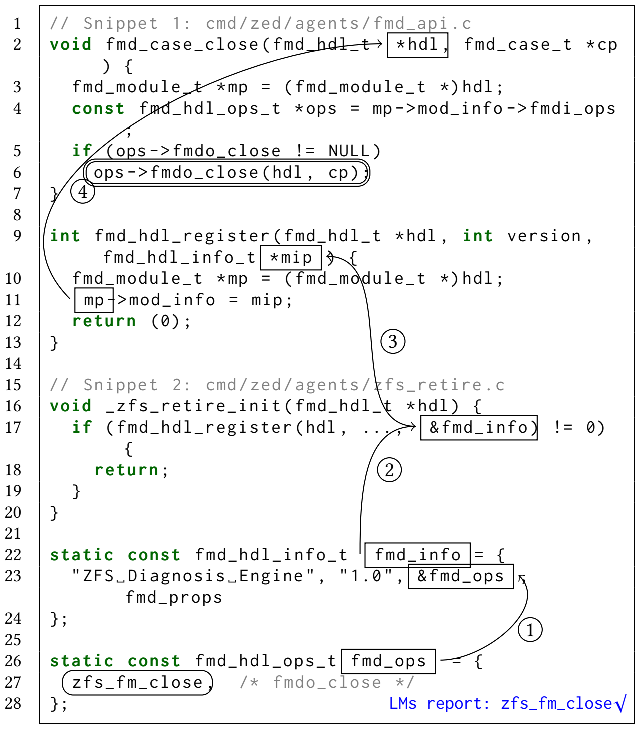
```
1  // Snippet 1: src/connection.h
2  int connSetWriteHandlerWithBarrier(connection *
       conn, ...) {
3    return conn->type->set_write_handler(...);
4  }
5
6  // Snippet 2: src/socket.c
7  static ConnectionType CT_Socket = {
8    .writev = connSocketWritev,
9    .read = connSocketRead,
10   .set_write_handler = connSocketSetWriteHandler,
11 };                   LMs report: connSocketSetWriteHandler√
```

Case 1. Call acrosses invisible code (from *redis*).



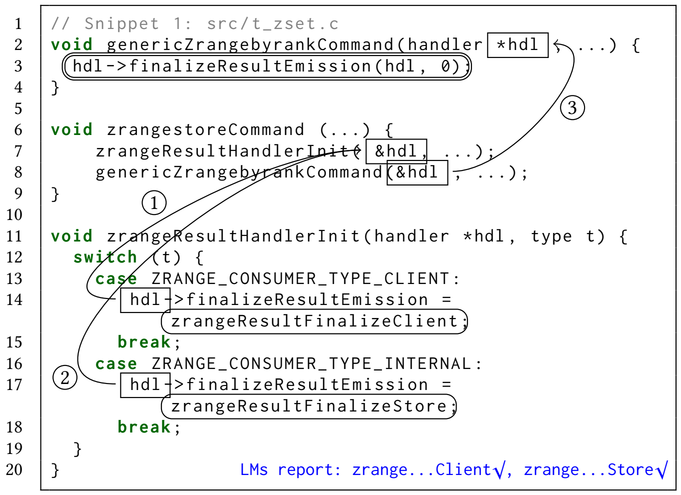
```
1  // Snippet 1: libavcodec/omx.c
2  void *dlsym_prefixed(void *hdl, char *sym, char *
       prefix) {
3    char buf[50];
4    snprintf(buf, sizeof(buf), "%s%s", prefix ? prefix:
         "", sym);
5    return dlsym(hdl, buf);
6  }
7
8  int omx_try_load(OMXContext *s, char *prefix) {
9    s->ptr_Init = dlsym_prefixed(s->lib, "OMX_Init",
         prefix);
10 }
11
12 OMXContext *omx_init(void *ctx, char *libname, char *
       prefix)
13 {
14   OMXContext *omx_context;
15   ...
16   omx_try_load(omx_context, ctx, libname, prefix, NULL
         );
17   omx_context->ptr_Init();
18   return omx_context;
19 }                                  LMs report: OMX_Init√
```

Case 3. Function name concatenated by strings (from *ffmpeg*).



```
1  // Snippet 1: cmd/zed/agents/fmd_api.c
2  void fmd_case_close(fmd_hdl_t *hdl, fmd_case_t *cp
       ) {
3    fmd_module_t *mp = (fmd_module_t *)hdl;
4    const fmd_hdl_ops_t *ops = mp->mod_info->fmdi_ops
         ;
5    if (ops->fmdo_close != NULL)
6    ops->fmdo_close(hdl, cp);
7  }
8
9  int fmd_hdl_register(fmd_hdl_t *hdl, int version,
       fmd_hdl_info_t *mip) {
10   fmd_module_t *mp = (fmd_module_t *)hdl;
11   mp->mod_info = mip;
12   return (0);
13 }
14
15 // Snippet 2: cmd/zed/agents/zfs_retire.c
16 void _zfs_retire_init(fmd_hdl_t *hdl) {
17   if (fmd_hdl_register(hdl, ..., &fmd_info) != 0)
         {
18     return;
19   }
20 }
21
22 static const fmd_hdl_info_t fmd_info = {
23   "ZFS␣Diagnosis␣Engine", "1.0", &fmd_ops,
         fmd_props
24 };
25
26 static const fmd_hdl_ops_t fmd_ops = {
27  zfs_fm_close, /* fmdo_close */
28 };                         LMs report: zfs_fm_close√
```

Case 2. Call relations hinted by comments (from *zfs*).



```
1  // Snippet 1: src/t_zset.c
2  void genericZrangebyrankCommand(handler *hdl, ...) {
3    hdl->finalizeResultEmission(hdl, 0);
4  }
5
6  void zrangestoreCommand (...) {
7      zrangeResultHandlerInit(&hdl, ...);
8      genericZrangebyrankCommand(&hdl, ...);
9  }
10
11 void zrangeResultHandlerInit(handler *hdl, type t) {
12   switch (t) {
13     case ZRANGE_CONSUMER_TYPE_CLIENT:
14       hdl->finalizeResultEmission =
             zrangeResultFinalizeClient;
15       break;
16     case ZRANGE_CONSUMER_TYPE_INTERNAL:
17       hdl->finalizeResultEmission =
             zrangeResultFinalizeStore;
18       break;
19   }
20 }         LMs report: zrange...Client√, zrange...Store√
```

Case 4. Function passed through simple data flows (from *redis*).

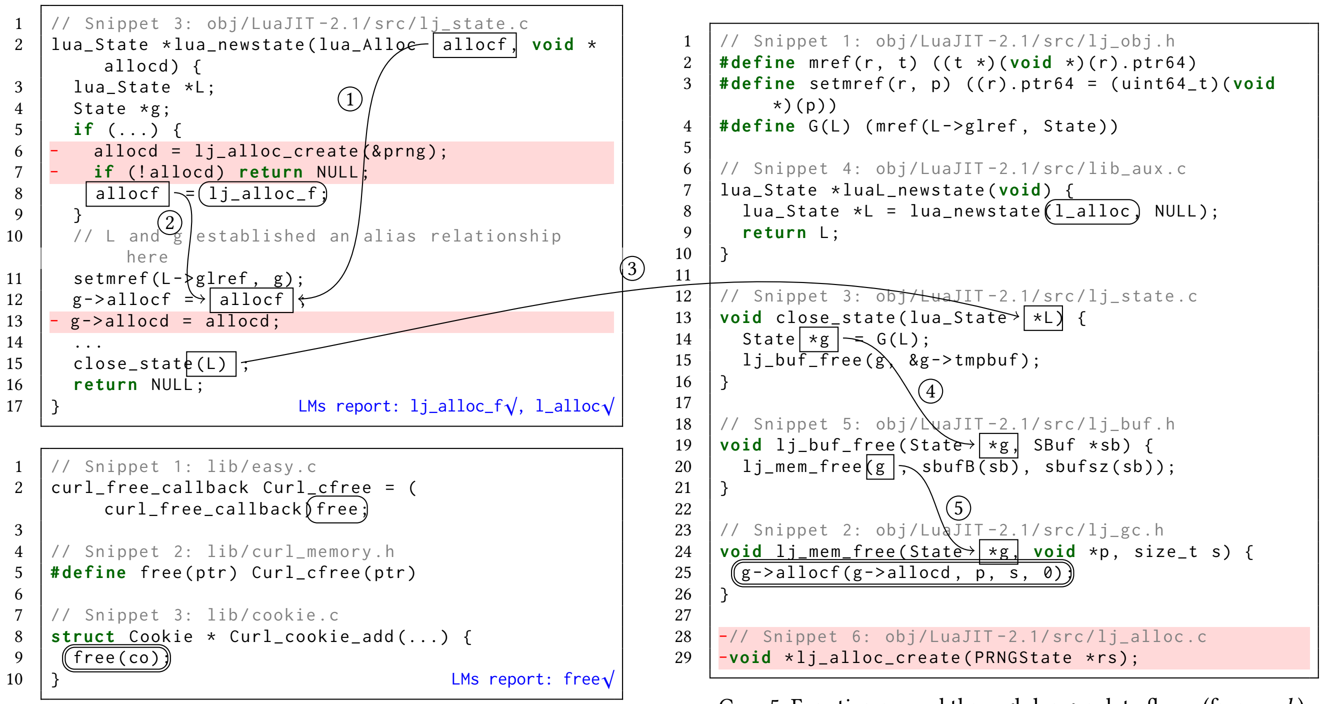
Fig. 1. Illustrative Case 1 - 4 in CG-Bench .

Case 1 illustrates a function pointer passes through black-box library functions. Finding the calling relationship with library functions is challenging as it usually requires whole program analysis or automatic function summarizes. It is hard to determine the value flow of the function pointer `set_write_handler` at line 3, when the library is large or its implementation is complex. However, in this scenario, **LMs can deduce the implicit value flow** from the initialization of `set_write_handler` at line 10.

Case 2 shows a function `zfs_fm_close` at line 27 and eventually be called by function pointer `fmdo_close` at line 6. For the sake of code readability, the developer annotates the call site of `zfs_fm_close` through comments. **LMs utilizes these comments** as well as the code snippets, infers correct indirect call targets. If this comment is omitted, CodeLlama-7b will report incorrect targets and Deepseek-6.7b will report additional false positive targets.

Finding 2: *LMs have certain understanding of data flows.*

Line 4 in *Case 3* takes the value of `prefix` and `sym` to concatenate the name of the target function. In traditional program analysis, it is challenging to not only models `dlsym` but also has to consider the result of such string computation [7, 9, 17]. **LMs seem to be good at string processing reasoning** and can infer the resulting values of `buf` from its calculation logic.



```
 1  // Snippet 3: obj/LuaJIT-2.1/src/lj_state.c
 2  lua_State *lua_newstate(lua_Alloc allocf, void *
        allocd) {
 3    lua_State *L;
 4    State *g;
 5    if (...) {
 6  -   allocd = lj_alloc_create(&prng);
 7  -   if (!allocd) return NULL;
 8     allocf = lj_alloc_f;
 9    }
10    // L and g established an alias relationship
          here
11    setmref(L->glref, g);
12    g->allocf = allocf;
13  - g->allocd = allocd;
14    ...
15    close_state(L);
16    return NULL;
17  }                LMs report: lj_alloc_f√, l_alloc√
```

```
 1  // Snippet 1: obj/LuaJIT-2.1/src/lj_obj.h
 2  #define mref(r, t) ((t *)(void *)(r).ptr64)
 3  #define setmref(r, p) ((r).ptr64 = (uint64_t)(void
        *)(p))
 4  #define G(L) (mref(L->glref, State))
 5
 6  // Snippet 4: obj/LuaJIT-2.1/src/lib_aux.c
 7  lua_State *luaL_newstate(void) {
 8    lua_State *L = lua_newstate(l_alloc, NULL);
 9    return L;
10  }
11
12  // Snippet 3: obj/LuaJIT-2.1/src/lj_state.c
13  void close_state(lua_State *L) {
14    State *g = G(L);
15    lj_buf_free(g, &g->tmpbuf);
16  }
17
18  // Snippet 5: obj/LuaJIT-2.1/src/lj_buf.h
19  void lj_buf_free(State *g, SBuf *sb) {
20    lj_mem_free(g, sbufB(sb), sbufsz(sb));
21  }
22
23  // Snippet 2: obj/LuaJIT-2.1/src/lj_gc.h
24  void lj_mem_free(State *g, void *p, size_t s) {
25    g->allocf(g->allocd, p, s, 0);
26  }
27
28  -// Snippet 6: obj/LuaJIT-2.1/src/lj_alloc.c
29  -void *lj_alloc_create(PRNGState *rs);
```

```
 1  // Snippet 1: lib/easy.c
 2  curl_free_callback Curl_cfree = (
        curl_free_callback)free;
 3
 4  // Snippet 2: lib/curl_memory.h
 5  #define free(ptr) Curl_cfree(ptr)
 6
 7  // Snippet 3: lib/cookie.c
 8  struct Cookie * Curl_cookie_add(...) {
 9    free(co);
10  }                          LMs report: free√
```

Case 6. Function pointer wrapped in a macro (from *curl*).

Case 5. Function passed through longer data flows (from *wrk*).

Fig. 2. Illustrative Case 5 and 6 in CG-Bench .

Case 4 shows indirect functions passing through simple data flows. In order to track indirect calls, LMs have to trace the structure handler along the data flows between functions, which demonstrates a capability similar to that of traditional program analysis tools.

Case 5 illustrates a more complex example of data flows. The alias relationship between g and L is established through macro setmref at line 11 in the left-hand side of *Case 5*. This example demonstrates the ability of LMs to trace data flow across branches, functions, and macros.

Finding 3: *Irrelevant information drops LMs' prediction performance.*

However, if irrelevant code marked by red background color is added to *Case 5*, LMs would then incorrectly report lj_alloc_create. The irrelevant data flow confuses LMs, especially when the irrelevant code shares similar variable names with the relevant part.

Finding 4: *Prompt hacking is able to improve inference performance.*

Case 6 shows an example that both the macro and the target function share the name free. This naming convention is deliberately chosen to enhance readability and portability. When applying two zero-shot prompt templates in the appendix, prompt (A.1) produces incorrect result Curl_cookie_add but prompt (A.2) produces correct result free. To figure out the prompt design for prompt (A.2), we tried different wording, involving different fields, and different orders of the fields. Table 2 shows, the improved zero-shot prompt leads to 5.23% improvement on average. Similar to other tasks [4, 30, 31, 33], prompt hacking will be one major source of performance improvement.

4 Conclusion and Future work

In this paper, we present CG-Bench , a benchmark for assessing LMs' proficiency in resolving indirect call targets in call graphs. We validate its efficacy via systematic evaluation of four prominent LMs; preliminary findings show notable effectiveness in this non-trivial task, highlighting their potential for call graph comprehension in program analysis. Future work will augment CG-Bench 's case repository with diverse paradigms, edge cases, and real-world artifacts for more rigorous evaluation of its discriminative power. Our objective remains establishing a comprehensive benchmark to holistically evaluate LMs' call graph understanding and advance software engineering research.

5 Acknowledgments

We appreciate the helpful comments from the anonymous reviewers, which significantly improved the quality of this paper. Thanks also to our colleagues Ge Zhang and Wenze Ma for stimulating discussions and ongoing support.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Anthropic. 2023. Model Card and Evaluations for Claude Models. (July 2023).

[3] Jeff Bonwick and Matthew Adams. 2001. ZFS: the last word in file systems. In *USENIX Conference on File and Storage Technologies*. 1–15.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *33nd USENIX Security Symposium (USENIX Security 24)*.

[6] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1126–1140.

[7] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. 2003. Precise analysis of string expressions. In *International Static Analysis Symposium*. Springer, 1–18.

[8] GNU Compiler Collection. 2024. GCC - The GNU Compiler Collection. https://gcc.gnu.org/

[9] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static analysis of string values. In *Formal Methods and Software Engineering: 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings 13*. Springer, 505–521.

[10] Al Danial. 2024. cloc: Count Lines of Code. https://github.com/AlDanial/cloc.

[11] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).

[12] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 72–82.

[13] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[14] William Gilbert. 2012. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. Accessed: 2024-01-23.

[15] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[16] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770* (2023).

[17] Evgeny Kudryashov. 2018. Static Analysis of Dlsym-Like Function Calls. In *2018 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 15–18.

[18] 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*. 1867–1881.

[19] FFmpeg Project. 2021. FFmpeg. https://www.ffmpeg.org/

[20] The OpenSSH Project. 2024. OpenSSH: Free Secure Shell (SSH) implementation. https://www.openssh.com/. [Online; accessed 23-January-2024].

[21] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[22] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[23] Salvatore Sanfilippo. 2009. Redis. https://redis.io/. Accessed: 2024-01-24.

[24] Ian Scheffler. 2023. GitHub CEO says Copilot will write 80% of code sooner than later. In *freethink*. https://www.freethink.com/robots-ai/github-copilot

[25] Daniel Stenberg. 1997. curl. https://curl.haxx.se/ Accessed: 2024-01-23.

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

[27] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[28] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[29] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[30] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[31] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2023. Large language models as optimizers. *arXiv preprint arXiv:2309.03409* (2023).

[32] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv preprint arXiv:2302.00288* (2023).

[33] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).

[34] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).

A Examples of LMs Input

In this section, we show some cases of model input (prompt) used in our experiment.

A.1 Zero-shot Prompt

Task: Given a segment of target code that includes a call chain and a function pointer, identify the exact name(s) of the real function(s) that the pointer according to the data flow refers to within the provided code. Do not invent or assume any functions that not appear in the code. If the function pointer could refer to multiple functions, list them separated by a semicolon. If there is no function that can be found, generate "None".
Important: Your response should only include the names of functions that are already defined in the given code. Do not generate any new function names. There can be multiple function names.
Target Code:

```
[CALL CHAIN CODE]
```

Function Pointer:
`[CALLSITE]`

Real Function Name(s):

A.2 Improved Zero-shot Prompt

Task: Determine the real function call name(s) corresponding to a given function pointer name in a code snippet containing a call chain according to the data flow refers to within the provided code. Do not invent or assume any functions that are not explicitly defined in the code. If there are multiple function names that the function pointer could refer to, separate them with a semicolon. If there is no corresponding function name, generate "None".
Input Format:

- Code snippet containing a call chain
- Function pointer name

Output Format:

- Real function call name(s) separated by a semicolon or "None".

Code Snippets:

```
[CALL CHAIN CODE]
```

Function Pointer Name:
`[CALLSITE]`

Output:

A.3 Few-shot Prompt

Task: Given a segment of target code that includes a call chain and a function pointer, identify the exact name(s) of the real function(s) that the pointer according to the data flow refers to within the provided code. Do not invent or assume any functions that do not appear in the code. If the function pointer could refer to multiple functions, list them separated by a semicolon. If there is no function that can be found, generate "None".
Important: Your response should only include the names of functions that are already defined in the given code. Do not generate any new function names. There can be multiple function names.
Few-Shot Examples:
Example 1:

```
static void (*f)(size_t) = func;

void *f1(size_t arg) {
  f(size);
}
```

Function Pointer:
`f`

Real Function Name(s):
`func`

Example 2:

```
static void func(handler *handler)
{
  switch (n) {
  case 1:
    handler->f = find;
      break;

  case 2:
    handler->f = lookfor;
      break;
  }
}
```

Function Pointer:
`handler->f`

Real Function Name(s):
`find; lookfor`

Target Code:

```
[CALL CHAIN CODE]
```

Function Pointer:
`[CALLSITE]`

Real Function Name(s):

B Other Examples in CG-Bench

B.1 Function pointers called after type casting

In *Case 7*, global variable `Curl_DIGEST_MD5` is an array of structures (with elements of type `MD5_params`, declared in line 6). When it's initialized in line 12, the member function pointer `md5_init_func` of the first element in the array

```
1   // Code snippet 1
2   #define CURLX_FUNCTION_CAST(target_type, func) \
3     (target_type)(void (*) (void))(func)
4
5   // Code snippet 2
6   struct MD5_params {
7     Curl_MD5_init_func  md5_init_func;
8   };
9
10  // Code snippet 3
11  const struct MD5_params Curl_DIGEST_MD5[] = {
12    { CURLX_FUNCTION_CAST(Curl_MD5_init_func,
          my_md5_init) }
13  };
14
15  // Code snippet 4
16  struct MD5_context *Curl_MD5_init(struct MD5_params
        *md5params) {
17    (*md5params->md5_init_func)(...)
18  }
19
20  // Code snippet 5
21  CURLcode Curl_auth_create_digest_md5_message(...) {
22      ctxt = Curl_MD5_init(Curl_DIGEST_MD5);
23  }
```

Case 7. Type casted function pointer (from *curl*).

```
1   // Code snippet 1
2   static log_handler_fn *log_handler;
3
4   void set_log_handler(log_handler_fn *handler) {
5     log_handler = handler;
6   }
7
8   // Code snippet 2
9   static void do_log(...) {
10    log_handler_fn *tmp_handler;
11    if (log_handler != NULL) {
12      tmp_handler = log_handler;
13      tmp_handler(...);
14      log_handler = tmp_handler;
15    }
16  }
17
18  // Code snippet 3
19  static int privsep_preauth(...) {
20    set_log_handler(mm_log_handler, ...);
21  }
```

Case 8. Function pointers involved in swapping (*openssh*).

```
1   // Code snippet 1
2   static void dump_object(...) {
3     if (...) {
4       object_viewer[ZDB_OT_TYPE(doi.doi_bonus_type)](...)
          ;
5     }
6   }
7
8   // Code snippet 2
9   static object_viewer_t *object_viewer[...] = {
10    dump_none,    /* unallocated */
11    ...
12    dump_unknown, /* Unknown type, must be last */
13  };
```

Case 9. Function within an array (from *zfs*).

```
1   // Code snippet 1
2   char * med3(..., int (*cmp) (const void *, const void
        *)) {
3     return cmp(a, b) < 0 ? ...;
4   }
5
6   // Code snippet 2
7   void _pqsort(..., int (*cmp) (const void *, const void
        *), ...) {
8     pl = med3(..., cmp);
9   }
10
11  // Code snippet 3
12  void pqsort(..., int (*cmp) (const void *, const void
        *), ...) {
13    _pqsort(..., cmp,...);
14  }
15
16  // Code snippet 4
17  void georadiusGeneric(...) {
18    int (*sort_gp_callback)(const void *a, const void *b)
          = NULL;
19    if (...) {
20      sort_gp_callback = sort_gp_asc;
21    } else if (...) {
22      sort_gp_callback = sort_gp_desc;
23    }
24    pqsort(..., sort_gp_callback, ...);
25  }
26
27  // Code snippet 5
28  void sortCommandGeneric(...) {
29    pqsort(..., sortCompare, ...);
30  }
```

Case 10. Function passed through callbacks (from *redis*).

Fig. 3. Other representative cases in CG-Bench .

is initialized to `my_md5_init`. What sets it apart is that a `void` type casting is used during this initialization process (through a macro defined in line 2). `Curl_DIGEST_MD5` is then passed as an argument to `Curl_MD5_init` in line 22, and eventually undergoes an indirect function call in line 17. In this scenario, in order to control the size of the possible set of target functions, MLTA considers pointers cast to `void` as escaped, hence failing to identify the possible set of indirect function calls at line 17, while SVF and LMs can produce correct reports.

B.2 Function passed through data flows

In *Case 8*, `log_handler` is a global function pointer defined at line 2. It is initialized as `mm_log_handler` at line 5 (in line 20, `mm_log_handler` is passed as an argument to the function `set_log_handler` function defined on line 4). In `do_log` function at line 9, `tmp_handler` declared on line 10 uses the value of `log_handler` and an indirect function call is made at line 13. In this example, `mm_log_handler` is passed between global and local function pointers. LMs report the correct answer in this scenario, which can also be easily resolved by MLTA and SVF.

B.3 Function from multiple sources

In *Case 9*, the global function pointer array `object_viewer` defined on line 9, contains over 50 functions including functions `dump_none` and `dump_unknown`. These functions are indirectly invoked through `object_viewer` on line 4. At the same line, `ZDB_OT_TYPE` is a macro that expands into a series of conditional statements, allowing for the selection of an index related to `doi.doi_bonus_type` at runtime. For the sake of conservative estimation in static analysis, all functions defined within `object_viewer` could potentially be called here. However, LMs can only output partial targets, while MLTA and SVF are capable of outputting all targets.

B.4 Callback across multiple functions

In *Case 10*, five distinct functions are defined, wherein `md3` (defined in line 2), `_pqsort` (defined in line 7), and `pqsort` (defined in line 12) are designed to pass a comparison callback function through parameters `cmp`. The other two remaining functions, `georadiusGeneric` (defined in line 17) and `sortCommandGeneric` (defined in line 28), respectively assign the actual function pointed to by `cmp` (through the `pqsort` function). The key difference lies in the function `sortCommandGeneric` directly passes the specific function `sortCompare`, while `georadiusGeneric` passes functions `sort_gp_asc` and `sort_go_desc` through the local function pointer `sort_gp_callback`. In such scenarios, MLTA and SVF can produce correct results, but LMs' result depends on how the prompt is conducted.

Received 2025-06-28; accepted 2025-08-08