RECOPILOT: REVERSE ENGINEERING COPILOT IN BINARY ANALYSIS

Guoqiang Chen Huiqi Sun Daguang Liu Zhiqi Wang Qiang Wang
Bin Yin Lu Liu Lingyun Ying
{guoqiangchen, sunhuiqi, liudaguang, wangzhiqi, wangqiang10,
binyin, liulu01, yinglingyun}@qianxin.com
QI-ANXIN Technology Research Institute, Beijing, China

June 13, 2025

ABSTRACT

Binary analysis plays a pivotal role in security domains such as malware detection and vulnerability discovery, yet it remains labor-intensive and heavily reliant on expert knowledge. General-purpose large language models (LLMs) perform well in programming analysis on source code, while binary-specific LLMs are underexplored. In this work, we present ReCopilot, an expert LLM designed for binary analysis tasks. ReCopilot integrates binary code knowledge through a meticulously constructed dataset, encompassing continue pretraining (CPT), supervised fine-tuning (SFT), and direct preference optimization (DPO) stages. It leverages variable data flow and call graph to enhance context awareness and employs test-time scaling to improve reasoning capabilities. Evaluations on a comprehensive binary analysis benchmark demonstrate that ReCopilot achieves state-of-the-art performance in tasks such as function name recovery and variable type inference on the decompiled pseudo code, outperforming both existing tools and LLMs by 13%. Our findings highlight the effectiveness of domain-specific training and context enhancement, while also revealing challenges in building super long chain-of-thought. ReCopilot represents a significant step toward automating binary analysis with interpretable and scalable AI assistance in this domain.

1 Introduction

Binary analysis plays a vital role in cybersecurity, empowering security professionals to uncover vulnerabilities, detect backdoors, and identify malware effectively. However, the lack of symbolic information in stripped binaries poses a significant challenge in understanding the binary code and performing analysis. Traditional decompilers, such as IDA Pro [1] and Ghidra [2], are powerful tools for reverse engineering and could generate C-like pseudo code, but they often struggle with predicting missing symbols, such as variable name and type, which are crucial for understanding the code's functionality. The decompiled pseudo code without source-level symbols is still difficult to read and understand, making it challenging for security analysts to reach the functional semantics efficiently.

General-purpose large language models (LLMs) have shown remarkable capabilities in various programming tasks, including code generation [3, 4, 5] and bug fixing [6, 7, 8]. A recent investigation study [9] has demonstrated that LLMs performed close to top-tier human competitors in competitive programming challenges. Considering the success of LLMs in the source-code domain, it is intuitive to explore their potential in binary analysis. The recent studies [10, 11, 12] have shown that LLMs can be effectively utilized to inferring symbols or reconstructing source code to assist the analysts. However, these efforts are still focusing on single task, such as decompilation [10] or variable name/type prediction [11, 12], leading to their lack of application potential in practice scenarios.

In this paper, we present an expert LLM, named ReCopilot, which is designed to be a *R*everse *E*ngineering *COPILOT* in binary analysis with multiple tasks supported, including decompilation, function name recovery, variable name/type recovery, struct recovery, and binary code summarization, etc. ReCopilot is trained on domain-specific data and tuned for equipping test-time scaling capability, which allows it to take a deep reasoning to improve the accuracy of the final

prediction. To build the model, we constructed three datasets to launch different training stages, including continue pretraining (CPT), supervised fine-tuning (SFT), and direct preference optimization (DPO). We also emitted a static program analysis on decompiled pseudo code to enhance the model's understanding of the code context and variable usages. Beyond the model building, we also developed a comprehensive benchmark to assess the performance of ReCopilot on various binary analysis tasks. The results demonstrated that ReCopilot achieved comparable performance to advanced general LLMs (DeepSeek-V3 671B) with a much smaller model size (7B), and outperforms the existing domain models by 13% averaged across tasks. To facilitate the security community, we have made the demo of ReCopilot publicly accessible to promote further research in this area[1].

2 Background

The source code is generally compiled into low-level machine code and assembled into executable files or runtime libraries, called binaries, which can be loaded into memory and executed by the CPU. The modern decompiler could lift machine code in bit stream into a high-level representation, *i.e.*, C-like pseudo code as shown in Figure 1a. Unfortunately, the debug symbols are often stripped before releasing the binaries for various purposes such as reducing file size and hiding functionality. The missing symbolic information, such as function names, variable names, and type definitions, makes it difficult to analyze the binaries. The traditional decompilation tools struggle with recovering these information, instead using placeholders (*e.g.*, `a1` and `v1`) in the pseudo code.

```
1  void __fastcall sub_1909(__int64 a1, __int64 a2, unsigned
   ↪  __int64 a3)
2  {
3    _OWORD *v3; // rcx
4    unsigned __int64 i; // rbx
5    _OWORD *v7; // rbp
6    __int64 j; // rax
7    __int64 v9; // rdi
8
9    v3 = (_OWORD *)(a1 + 176);
10   for ( i = 0LL; ; i += 16LL )
11   {
12     v7 = (_OWORD *)(a2 + i);
13     if ( i >= a3 )
14       break;
15     for ( j = 0LL; j != 16; ++j )
16       *((_BYTE *)v7 + j) ^= *((_BYTE *)v3 + j);
17     v9 = a2 + i;
18     sub_14F7(v9, a1);
19     v3 = v7;
20   }
21   *(_OWORD *)(a1 + 176) = *v3;
22 }
```

(a) Pseudo Code

```
1  #define AES_keyExpSize 176
2  #define AES_BLOCKLEN 16
3  struct AES_ctx
4  {
5    uint8_t RoundKey[AES_keyExpSize];
6    uint8_t Iv[AES_BLOCKLEN];
7  };
8  void AES_CBC_encrypt_buffer(struct AES_ctx *ctx, uint8_t*
   ↪  buf, size_t length)
9  {
10   size_t i;
11   uint8_t *Iv = ctx->Iv;
12   for (i = 0; i < length; i += AES_BLOCKLEN)
13   {
14     XorWithIv(buf, Iv);
15     Cipher((state_t*)buf, ctx->RoundKey);
16     Iv = buf;
17     buf += AES_BLOCKLEN;
18   }
19   /* store Iv in ctx for next call */
20   memcpy(ctx->Iv, Iv, AES_BLOCKLEN);
21 }
```

(b) Source Code

Figure 1: An example of decompiled pseudo code and the corresponding source code from a real-world `AES` implementation [13].

Figure 1 shows an example of pseudo code decompiled by IDA Pro and its corresponding source code, which is an AES encrypt function from an open source repository [13]. It is difficult to identify the high-level semantics of the pseudo code function even though it exhibits a syntactic structure similar to the source code. As for the source code, we can easily make a determination even with just the function name, but unfortunately, these symbols are erased from the stripped binaries. On the other hand, binary analysis tasks, like vulnerability detection, often require a deep understanding of the data structure of a variable, which is often lost in the decompiled code. Based on the source code, it is easy to tell that the first parameter of the function shown in Figure 1 accepts a struct variable of type `AES_ctx`, but such conclusion is not straightforward in the decompiled code. We can only infer from `L9` and `L16` that the position at an offset of 176 from variable `a1` is a 16-byte length data. However, to determine the specific structure of the first 176

[1] `https://tqgpt.qianxin.com/recopilot`

bytes of `a1`, more information needs to be gathered by analyzing both the caller and callee of `sub_1909`. In general, it is much harder to understand the functionality of the decompiled pseudo code, since it has no meaningful symbol.

In recent years, LLMs have shown great potential in understanding, generating, and analyzing programs, providing substantial assistance to programming tasks in the source code. EvalPlus [14, 15] hosts a leaderboard for evaluating the performance of LLMs on code synthesis. As of the completion of this paper, EvalPlus with HumanEval [16] dataset have been almost fully passed by the top models, with o1-preview achieving a score of 96.3. SWE-bench [17] is an evaluation framework consisting of real GitHub issues, designed to measure the performance of LLMs in resolving codebase defects. According to the SWE-bench leaderboard, the SOTA method driven by Claude 3.7 Sonnet has successfully resolved 33% of the issues in 2025, highlighting a significant and rapid improvement compared to the 2% achieved by the best-performing model in 2023.

Given the success of LLMs in source code, it is intuitive to prompt general LLMs to perform binary code analysis, such as vulnerability detection and symbolic information recovery. There are several investigation studies that show the potential and limitation of these LLMs in binary analysis tasks. BinSum [18] present a large-scale assessment of LLMs on binary code summarization, and show that they can generate high-quality summaries for binary functions, however, the performance is dropping significantly facing stripped binaries. Another work [19] also evaluate the LLMs in binary code understanding with function name prediction and binary code summarization tasks, and drew similar conclusions. DeBinVul [20] employs vulnerability detection, classification, and summarization tasks for evaluation, revealing the sub-optimal performance of general models compared to the fine-tuned domain models. In general, we consider binary analysis as a domain-specific problem, where general-purpose LLMs, without specialized training, still struggle to reach the level of human experts. In other words, developing expert models presents a promising direction for integrating AI technologies into the field of binary program analysis.

Related Work. There are several attempts to predict the symbolic information of binaries using neural-based techniques. The following research efforts [21, 22, 23, 24, 25, 26, 27] are designed to predict the source-level names of binary functions to facilitate understanding by analysts. There are also several studies that focus on predicting the variable names [28, 29, 30, 31, 12] and types [28, 32, 33, 34, 35, 11] in the decompiled pseudo code by developing neural network-based approaches, and some of them, such as TYGR [34] and ReSym [11], attempt to predict the memory structure of complex variables (*e.g.*, struct). Meanwhile, others make efforts to fine-tune models to learn code semantics and generate comprehensive summaries [36, 31] even source code [31, 10] for binary functions. All of these studies have achieved promising results in their respective evaluations and offered valuable insights for subsequent research. However, they are limited to addressing only one specific task, rendering their methods insufficient for meeting the diverse demands of real-world binary analysis. On the other hand, only a few studies, like LLM4Decompile [10], attempt to leverage LLMs for binary analysis tasks, which leaves a lot of room for our research.

Challenges. The easiest approach is to prompt existing LLMs to perform binary analysis tasks. As mentioned above, however, previous evaluations [18, 19, 20] have demonstrated that it yields limited performance. Therefore, we here aim to train an expert LLM to support the most important and common tasks in binary analysis practice. To this end, we need to address several challenges:

1) Lacking publicly available datasets, we need to collect large-scale and fine-grained domain data, specifically align data between stripped binaries and source code.

2) Considering the offline analysis requirements in practice, we expect a small LLM to support local inference, which should be laptop-deployable. This motivates us to train the model carefully to prevent overfitting while ensuring good performance.

3) The context limitation of LLMs constrains us to analyze only a portion of a large binary program at one time. The previous methods typically make the prediction based on a single binary function, resulting in providing insufficient information. Therefore, we aim to enable the model to utilize context information to improve analysis, which requires additional efforts in context building.

3 Methodology

3.1 Overview

Figure 2 shows the overview of our model building process that mainly consists of three training stages. Specifically, we first conduct continued pretraining (CPT) on a pretrained base model to learn domain language and knowledge. Then, a supervised fine-tuning (SFT) is performed with a mixture SFT dataset to empower the model with reasoning ability and adapt it to the downstream tasks. Finally, we employ the direct preference optimization (DPO) to further improve the model's performance in the format-following. We will introduce the details of each stage in §3.2.

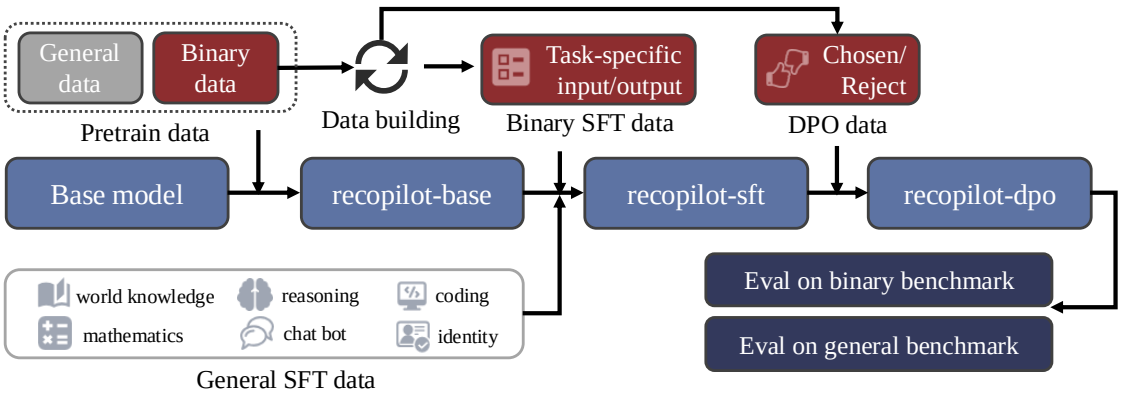

Figure 2: An overview of the ReCopilot model building.

We have built corresponding datasets from scratch to launch these trainings. Several pipelines are firstly designed to collect the raw binary dataset, which is then filtered and sanitized to build the final training datasets. To build a task-specific SFT dataset for binary analysis, we devise a generator-discriminator framework to automatically generate domain examples with chain of thought. In general, we build a large-scale pretrain dataset with 60B tokens, a mixture SFT dataset with 1.7B tokens, and a DPO dataset with 2.4K examples, and they will be detailed in §3.3.

We also propose a context enhancement method to improve the prompt building for the target binary function. The context enhancement employs static program analysis to build function context, call chains, and variable data flow, which are organized into the model input to provide as comprehensive a view as possible. We will present the whole prompt specification and more building details in §3.4.

3.2 Training Strategy

CPT. Pretraining is generally used to acquire foundational knowledge and learn language features from large-scale data, enabling efficient adaptation to diverse downstream tasks through transfer learning. General-purpose LLMs [37, 38, 39] are pretrained on wide-ranging corpora, such as books, Wikipedia, and web pages, which rarely contain binary code (*i.e.*, decompiled pseudo code). And the publicly exposed pseudo code often lacks corresponding source code information, indicating a low data quality. These facts suggest that the general LLMs are underfitting to the binary domain.

To better model pseudo code, we conducted continued pretraining (CPT) on a base LLM using a customized dataset. Meanwhile, we took further efforts to make the model learn the mappings between binary code, source code, and natural language. Overall, our CPT injected domain knowledge into the model and improves its understanding of binary code, which is crucial for the subsequent fine-tuning on downstream tasks.

SFT. We employed supervised fine-tuning (SFT) to adapt the pretrained model (*i.e.*, recopilot-base in Figure 2) to the pre-defined tasks for binary analysis. SFT enables the model to follow user instructions and generate results in specific formats, which is essential for our downstream applications. Furthermore, we consider binary analysis tasks generally are reasoning-intensive, like solving math problems, indicating that the model needs to learn how to reason about code semantics and generate accurate results. Inspired by OpenAI-o1 [40] and DeepSeek-R1 [41], we plan to equip our model with the test-time scaling ability. To this end, we propose a generator-discriminator framework (detailed in §3.3.3) to automatically generate SFT data with Chain-of-Thought (CoT), enabling us to fine-tune our model to take a deep thinking before giving the answer. Finally, our recopilot-sft model accepts a prompt built from a decompiled binary, and generates a reasoning process for the user-specific task following with the final prediction in JSON format, which is easy to parse and apply into decompilation tools.

DPO. We further deployed direct preference optimization (DPO) [42] to improve the recopilot-sft's performance in format-following and reasoning consistency. Format errors directly affect the usability of the model, and rigorous logic is crucial for generating correct reasoning. Specifically, DPO is a reinforcement learning method that optimizes the model's output by directly learning from user preferences, involving data pairs that consist of chosen and rejected responses for the same prompt. In contrast to traditional RLHF [43] methods, it does not need human feedback and a complex reward function, and allows us launch the training efficiently on limited computation resources.

3.3 Dataset Building

3.3.1 Raw Dataset Collection

To build a large-scale, fine-grained raw dataset of binary functions, we start with numerous stripped binaries and source code packages. In practice, we identified different data sources and designed several automatic pipelines to collect the data. Specifically, we have built the following pipelines:

- **1) Compile from Scratch:** We pull source code from the online package repositories, such as Archlinux [44], which host a large number of open-source software packages and organize them in a well-structured manner, enabling batch compilation with specific arguments. We further utilized a compiler wrapper to precisely control compilation options to building stripped binaries and corresponding debug information. Although the compilation process often fails for various reasons, such as missing dependency libraries, we still collected a large amount of data from this pipeline, specifically ≈60 million binary functions.
- **2) Off-the-shelf Software Artifacts:** To enrich the dataset, we also collected binaries from existing software artifacts repositories, such as the Ubuntu [45] and Debian [46] package sources. One triplet is clustered by the project name and version, and consists of a release package, a debug symbol package, and a source code package. These software packages are built from various compilation environments far beyond the compilers and options we used in the "Compile from Scratch" pipeline, which helps us to cover a wider range of real-world programs.
- **3) CompileAgent:** To further include ad-hoc projects, we employed a recent work, CompileAgent [47], to automatically building project from source code. CompileAgent is a LLM-agent driven framework that can take over compilation from a specific repository URL or a local codebase and handle the possible errors during the whole process. It mitigates the limitation of handling compilation errors automatically in other pipelines, providing us with critically needed projects without more manual efforts.

We obtained large-scale binary files and corresponding source code packages from these pipelines. To connect the binary functions to their source code counterparts, we leverage the debug information that contains the file path and line number for each function definition. The binary functions are decompiled by the modern decompiler IDA Pro [1], and the source function are extracted by the programming parser tree-sitter [48]. For the current version, we mainly focus on C/C++ binaries and source code.

Sanitization and Deduplication. Data noise could bias the model and lead to overfitting. To improve the data quality, we performed a series of sanitization and deduplication steps. First, we removed the binary function with too short/long length, which could contain insufficient or excessive information to prevent effective learning. The thunk functions only consist of jump instructions to forward calls to other functions, and the auxiliary functions (*e.g.*, `register_tm_clones`) are generated by the compiler for assisting the program execution, we thus filtered out these functions to reduce noise. Moreover, we also removed the functions with no source code found, *i.e.*, missing the ground truth, which mostly correspond to third-party library code introduced in the binaries. On the other hand, there are many similar pseudo code functions in the dataset, which could be due to the reusing code across projects. To reduce the redundancy, we applied the MinHash [49] algorithm to perform function-level deduplication on our raw dataset. MinHash is a locality-sensitive hashing algorithm that can efficiently identify similar items in large datasets.

In general, our raw dataset consists of 101M binary functions collected from 11K projects, which is far beyond the 100K functions collected in previous work LLM4Decompile [10], and the detailed statistics are shown in Table 1:

Table 1: Statistics for our raw dataset in the binary domain.

Pipeline	# Project	# Binary	# Function
Compile from Scratch	4,350	506,138	59,927,927
Off-the-shelf Software Artifacts	7,021	340,029	40,750,991
CompileAgent	101	9,733	880,414
In Total	**11,472**	**855,900**	**101,559,332**

3.3.2 Pretraining Dataset

We sampled a large-scale domain pretraining dataset from the raw dataset, which is used to learn the data representation and build connections across binary, source, and natural language. To this end, as shown in Figure 3, each sample is constructed by a stripped binary function in pseudo code, the corresponding source code, and a comment in natural language. We also included the decompiled pseudo code with debug symbols in each sample, facilitating the model

to learn the binary code better. Meanwhile, we involved the definitions for complex types (*i.e.*, struct and enum) in the source code part. LLM learns the material in pretraining through the next-token prediction task, which means learning to predict subsequent tokens from preceding token sequences in a piece of data. In order to build a bidirectional mapping of different data modals rather one way, we took inner-shuffling to disorder three segments in one single data.

Figure 3: An example for demonstrating the pretraining data format and inner-shuffling.

To prevent our model from overfitting on binary data, we further collected general text and code data from the open-source datasets. Specifically, we randomly sampled C/C++/Python/Rust/Go/Shell code from The Stack v2 [50] and RedPajama [51] datasets. Regarding the natural language data, we sampled from the following high-quality datasets: wikipedia [52], stackoverflow-posts [53], and security-paper-datasets [54], where only the English and Chinese texts are selected, and we prioritize reverse-engineering related documents for sampling.

Being focused on building an expert LLM, we proactively make our binary code data dominant in the final pretraining dataset. With references to previous work [55, 56], we set the mixture ratio of binary/code/text data to 60:25:15. In summary, our final pretraining dataset contains 36B tokens, including 21B tokens from binary domain, 10B tokens from general code, and 5B tokens from natural language. The detailed statistics are shown in Table 2.

Table 2: Statistics of our pretraining (PT) dataset.

Data Source	Domain	# Samples	# Tokens(B)
the-stack-v2 [50]	code	4,805,445	5.58
RedPajama [51]	code	4,343,832	4.08
wikipedia [52]	text	3,593,584	2.04
security-paper-datasets [54]	text	428,155	0.53
stackoverflow-posts [53]	text	3,964,004	2.48
binary-raw-dataset	binary	5,733,356	21.18
In Total	/	**20,233,101**	**35.83**

3.3.3 Generator-Discriminator Framework for SFT Dataset

To build the binary analysis SFT dataset, we first identified 14 tasks that are commonly used in the field and the most helpful for the participants in reverse engineering. As mentioned in §2, the related works are mainly focused on function name recovery, variable name/type prediction, and binary code summarization, which are generally used to assist binary code understanding and considered as the most important tasks in this field. Based on these primitive tasks, we further defined the following 14 specific tasks to fine-tune our base model. The following is the task tags and descriptions:

- **Function Name Recovery:** `<funcname>` Given a function in decompiled pseudo code, recovery the function name in the source-code level.
- **Function Signature Recovery:** `<signature>` Given a function in decompiled pseudo code, recovery the signature (i.e., function definition) into the source-code level.
- **Variables Recovery:** `<vars>` Given a function in decompiled pseudo code, recovery the variables into the source-code level, including the variables' types (including structs) and names.
- **Arguments Recovery:** `<args>` Given a function in decompiled pseudo code, recovery the arguments into the source-code level, including the arguments' types (including structs) and names.
- **Variable Recovery:** `<var:var_name>` Given a function in decompiled pseudo code, recovery the specific variable into the source-code level, including the variable' type (including struct) and name.
- **Argument Recovery:** `<arg:arg_name>` Given a function in decompiled pseudo code, recovery the specific argument into the source-code level, including the argument' type (including struct) and name.

- **Algorithm Identification:** `<algorithm>` Given a function in decompiled pseudo code, identify whether this function is a particular algorithm, or part of its implementation.
- **Category Identification:** `<category>` Given a function in decompiled pseudo code, identify the functionality category of this function.
- **Brief Summary:** `<summary-brief-en>, <summary-brief-cn>` Given a function in decompiled pseudo code, generate a brief summary (1-2 sentences maximum) of the function in natural language.
- **Detailed Summary:** `<summary-en>, <summary-cn>` Generate a detailed summary in English describing the function's purpose, arguments, return value, functionality category, and possible algorithm, as well as inline comments if should have.
- **Binary Function Analysis:** `<func-analysis>` Given a function in decompiled pseudo code, conduct a detailed analysis to recovery the following information: return type, function name, arguments, variables, detailed comments, functionality category and possible algorithm.
- **Decompilation:** `<decompilation>` Given a function in decompiled pseudo code, improve the pseudo code, make it closer to source code and more understandable.

Notably, the binary function analysis task tagged by `<func-analysis>` is an overall analysis for the binary function user asked and almost covers all of the other tasks. However, we still involve multiple independent tasks to fine-tune our model rather only the one, primarily due to the transferability between tasks, where different tasks exhibit mutual benefits. For example, improvements in type inference can potentially enhance the model performance in the semantic understanding task and vice versa.

We carefully defined a general input-output template for the binary analysis tasks, as shown in Figure 4. The template is designed to be modular and flexible, allowing for scalability to accommodate more tasks. Specifically, the template is composed of the following parts: ① targeted binary function (*i.e.*, pseudo code), ② context functions, ③ call chains, ④ data flow, ⑤ task tag, ⑥ model reasoning process, and ⑦ the final prediction. The ① - ⑤ colored in red are assembled to the whole model input, while the ⑥ and ⑦ colored in purple are the model output. The input includes not only the target function but also the static program analysis results (*i.e.*, ②③④), which are considered as the context enhancement being helpful for model analysis, and we will present the details later in §3.4. The ground truth is obtained from source code bridged by debug information (*e.g.*, DWARF); we then reformat them into specific JSON format as the final prediction. However, there is still a critical challenge in building the SFT dataset: the lack of thinking process in the model output.

```
<context-pseudocode>
{context}
</context-pseudocode>
<pseudocode>
{target_func}
</pseudocode>
<Call-Chains>
{call_chains}
</Call-Chains>
<Data-Flow>
{data_flow}
</Data-Flow>
Analysis Task Tag:
{task_tag}
<Thought>

Thinking...
</Thought>
<Output>JSON Format</Output>
```

Figure 4: The input-output template designed for binary analysis tasks in ReCopilot.

Since we aim to build a reasoning model that can take a deep thinking for the user-specific task, we need to provide the model with a large number of examples that contain the reasoning process. It is an intuitive idea to distill existing reasoning LLMs (*e.g.*, DeepSeek-R1) and use reject sampling to collect such dataset. Unfortunately, the existing LLMs are not well-trained on binary code and often produce incorrect reasoning processes, leading to mispredictions. Moreover, if one incorporates ground truth into the prompt, the reasoning process of these LLMs tends to refer to them directly, and thus cannot be used as training data.

To solve this problem, inspired by previous work in data synthesis [57, 58], we proposed a generator-discriminator framework to automatically generate SFT dataset with reasoning processes for binary analysis tasks, as shown in Figure 5. We first collect the raw SFT data from our raw dataset, which contains the input, output without CoT, corresponding source code, and meta information (*e.g.*, file name and project name). Then, we assemble the raw SFT data and the generation guide into the generator prompt, which is used to prompt a general LLM (*e.g.*, DeepSeek-V3) to generate a CoT without any direct citations to the ground truth. A format-based G-Parser is developed to extract the generated CoT that is subsequently filled into the discriminator prompt. The discriminator is also driven by a general LLM to judge whether the generation meets our requirements. The requirements mainly involve correctness, consistency, helpfulness, and purity. A similar D-Parser is used to access the discriminator's judgment to determinate

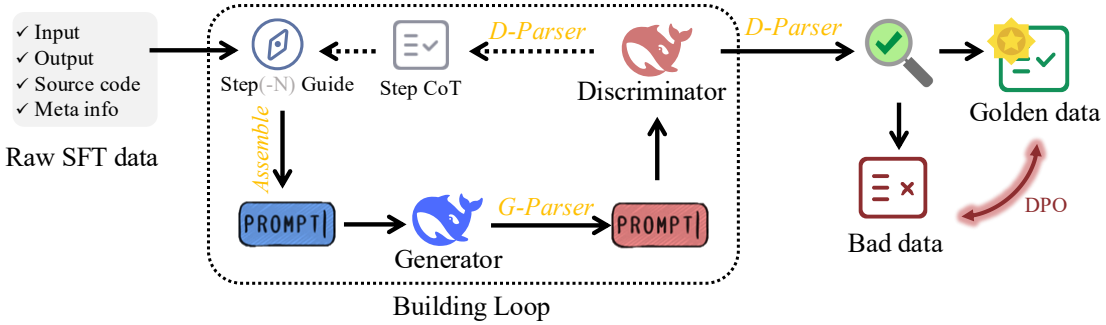

Figure 5: An overview of the generator-discriminator framework for building supervised fine-tuning (SFT) data of binary analysis tasks. The dashed line arrows indicate an optional building loop that generates a single-step chain-of-thought (CoT) each time.

the destination of the generated CoT. In the above program, we invoked the LLM twice to generate one SFT data with a reasoning process, which is efficiently building with a success rate of more than 90%. However, we found that these CoTs are generally short with an average of 876.33 tokens, especially compared to DeepSeek-R1's thinking.

Super-CoT. To further unleash the power of test-time scaling, we employed a step-by-step building loop to construct super-long chains-of-thought (called Super-CoT), as shown in the dashed box in Figure 5. Benefiting from the limitation of our problem scope to the binary domain, we can clearly define the reasoning steps for each domain task and thus perform stepwise generation, which cannot be practiced on general reasoning tasks. In the building loop program, we assemble the raw SFT data and the expert-written step-N guide into the generator prompt to generate the CoT for the current step only. And the discriminator makes a judgment on the current step CoT. If only the current generation is qualified, we will continue to generate the next one with the previous step CoTs integrated. In this way, we invoked the LLM multiple times to generate one SFT data with a Super-CoT, resulting in ≈10 times longer reasoning process than before. To distinguish it from the usual reasoning process, we modified the thinking tag at the end of the model input to `<Super-Though>` for enabling the model to conduct Super-CoT. Nevertheless, this program significantly increases the overhead and reduces the overall success rate, making it costly to scale the Super-CoT dataset.

Beyond the binary analysis SFT dataset, we also sampled general SFT data from open-source datasets, especially the reasoning data in code and math domains. Previous studies [41, 59, 60] present empirical evidence that reasoning ability is transferable from task to task. For example, the math reasoning ability can be transferred to code reasoning tasks, and vice versa. Finally, we built the SFT dataset with 403K examples, detailed in Table 3.

Table 3: Statistics of our supervised fine-tuning (SFT) dataset.

Data Source	Category	# Samples	# Tokens(B)
tulu-3-sft-mixture [58]	mixture	50,000	0.0397
OpenHermes-2.5 [61]	mixture	20,000	0.0079
WizardLM_evol_instruct_V2_196k [62]	instruct	20,000	0.0103
OpenMathInstruct-2 [63]	math	10,000	0.0049
OpenO1-SFT [64]	reasoning	77,685	0.0922
OpenThoughts-114k [65]	reasoning	113,957	0.7993
recopilot-sft-cot	binary	99,461	0.6088
recopilot-sft-super-cot	binary	11,781	0.1565
Identity	identity	465	0.0001
In Total	/	**403,349**	**1.8356**

DPO dataset. During the construction of the SFT dataset, low-quality generations were inevitably observed, including issues such as formatting errors and logical inconsistencies. For such cases, we implement a retry mechanism for re-invoking generation with the same raw SFT data. And if the retry produces qualified data, as shown by the highlighted red line in Figure 5, the failure-success pair can be utilized as training data for DPO. A DPO training example consists of a chosen response and a rejected response to the same prompt, enabling the model to learn the preference for alignment. Through this approach, we reduced resource wastage by effectively leveraging otherwise unusable data. In total, we have constructed a dataset of 2.4K DPO samples.

3.4 Context Enhancement

```
<context-pseudocode>
0| __int64 sub_1ABB()
1| { ..... }
0| __int64 __fastcall sub_14F7(__int64 a1, __int64 a2, __int64 a3, __int64 a4, __int64 a5)
1| { ..... }
</context-pseudocode>
<pseudocode>
0| void __fastcall sub_1909(__int64 a1, __int64 a2, unsigned __int64 a3)
1| { ..... }
</pseudocode>
<Call-Chains>
sub_1ABB-->sub_1909
sub_1909-->sub_14F7
</Call-Chains>
<Data-Flow>
==== final usages for traced variable: `['a1']` in `sub_1909` ====
---- trace callee usages ----
sub_1909@L0||void __fastcall sub_1909(__int64 a1, __int64 a2, unsigned __int64 a3) // alias: __int64 a1 == a1
sub_1909@L17||sub_14F7(v9, a1); // alias: __int64 a1 == a1
       sub_14F7@L0||__int64 __fastcall sub_14F7(__int64 a1, __int64 a2, ..... // alias: __int64 a2 == a1
       ......
</Data-Flow>
Analysis Task Tag:
<arg:a1>
<Thought>
```

Figure 6: An example of prompt with context enhancement by ReCopilot. The variable aliases yield from data flow analysis are highlighted in yellow.

In this section, we present our context enhancement method designed to improve the prompt for analyzing target binary function. To provide a comprehensive view of the binary code, we employ static program analysis on pseudo code to collect context functions, call chains, and variable data flow. Specifically, we first present our call chain analysis (§3.4.1) that traverses the call graph to identify and select the most informative context functions. Then, we introduce our data flow analysis technique (§3.4.2) that traces variable propagation across functions to understand their usage patterns and relationships. As illustrated in Figure 6, these contextual elements are systematically organized in the model input.

3.4.1 Call Chain Analysis

Starting from the target binary function, we employ a breadth-first search (BFS) strategy to traverse the call graph constructed from decompiled pseudo code. The BFS traversal systematically explores both direct callers and callees of the current function, with the traversal depth limited by a user-specified parameter to prevent context explosion. The traversal terminates at leaf nodes (functions with no callees) or root nodes (functions with no callers) in the forward and backward directions respectively, as well as at previously visited functions to avoid cycles. Through this bidirectional traversal, we obtain a collection of call chains and their associated context functions. We organize the context functions' pseudo code in descending order based on their depth in the call graph, placing functions closer to the target function near the end of the prompt to leverage the model's stronger attention to recent context.

Informative Score Measurement. Involving too many functions into prompt can easily exceed the context length limitation. To address this challenge, we introduce an empirical metric that quantifies the information richness of a pseudo code function, which correlates with the function's comprehensibility to the LLM. This informative score is computed by analyzing three key components: the presence of meaningful function names, the density of string literals, and the semantic information from the callee names:

$$\mathcal{N}(f) = \begin{cases} 1, & \text{if the name symbol of } f \text{ exists} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$\mathcal{S}(f) = \mathcal{N}(f) + \min\left(1, \frac{\beta \cdot \text{num\_strings}(f)}{\text{num\_lines}(f)}\right) + \sum_{c \in \text{callees}(f)} \frac{\mathcal{N}(c)}{|\text{callees}(f)|} \tag{2}$$

where $\mathcal{S}(f)$ is the informative score of function f, $\mathcal{N}(f)$ is an indicator for meaningful function name, and the β is a scaling factor for string density measurement. We empirically set β to 25, means the function with 1 string in each 25 pseudo code line is considered as string-rich and informative.

Finally, we sorted the traversed context functions by their informative scores and selected the top-k functions to be included into the prompt. The parameter of k is configurable according to the model's max context length, which is set to 10 by default in our experiments. Moreover, in fact, we determine a limited number of context functions not only by the control flow analysis but also by increasing the priority of functions reached by the data flow, which is particularly important for tasks like variable type inference.

3.4.2 Data Flow Analysis

To analyze the source-code type of the variable in binary code, the all usage patterns should be inspected carefully throughout the call graph. We harnessed LLMs to reasoning about it, just like what human experts do. Unfortunately, it is difficult for LLMs to trace a variable's data flow in a bunch of pseudo code, especially when the variable is passed into multiple functions and through a deep call stack. To help LLMs with that, one should build easy-to-read variable usage patterns for LLMs.

We have investigated the existing data flow analysis tools, such as Joern [66] and Semgrep [67], which could potentially be integrated into ReCopilot. However, several obstacles prevent their adoption: 1) Although similar, pseudo code is not identical to the C/C++ language, and there is no official support for decompiled pseudo code in these tools; 2) The community version of Semgrep supports intra-procedural data flow only, which is not sufficient for our cross-function analysis needs; 3) Both of them are heavy-weight engines designed for software vulnerability discovery, introducing unnecessary computational overhead in our scenario. Therefore, we opt to implement our own lightweight data flow analysis solution to trace variables in decompiled pseudo code.

In practice, we trace variable propagations by recursively traversing the abstract syntax tree (AST) of the decompiled pseudo code. Beginning with the target function, we perform a depth-first traversal of the AST. For each variable node encountered, we execute the analysis based on predefined inference rules, detailed in Figure 7. To avoid too many confusing mathematical symbols, we use programming-like statements in the rules' definition rather than formal Hoare logic notations for better readability. The variable we are interested in is marked as traced at definition statements (Figure 7a), and will be passed to new variables within the same function by assignment statements (Figure 7c). We also record each expression using the traced variables (Figure 7b), which is helpful for the model to understand the variable's usage patterns. When encountering function calls, we recursively traverse the callee's AST following the rules defined for calling statements (Figure 7d), propagating traced variables through function arguments. For backward propagation analysis, we employ a breadth-first traversal of the target function's callers, propagating traced arguments into the caller contexts and applying the same analysis to each caller's AST (Figure 7e). Notably, we no longer handle the function callings in the callers to reduce the overhead.

$$\frac{\texttt{is\_traced(v), is\_def(v)}}{\texttt{set\_alias(v,v), log\_usage(v)}} \quad (3)$$

(a) Definition Statement

$$\frac{\texttt{is\_traced(v), in\_expr(v)}}{\texttt{log\_usage(v)}} \quad (4)$$

(b) Expression Statement

$$\frac{\begin{array}{l}\texttt{is\_traced(v), in\_asg(v), in\_rvalue(v),}\\ \texttt{is\_simple(rvalue), is\_simple(lvalue)}\end{array}}{\begin{array}{l}\texttt{set\_alias(lvalue,get\_alias(rvalue)),}\\ \texttt{set\_is\_traced(lvalue), log\_usage(v)}\end{array}} \quad (5)$$

$$\frac{\begin{array}{l}\texttt{is\_traced(v), in\_asg(v), in\_lvalue(v),}\\ \texttt{is\_simple(lvalue), is\_simple(rvalue)}\end{array}}{\begin{array}{l}\texttt{set\_alias(rvalue,get\_alias(lvalue)),}\\ \texttt{set\_is\_traced(rvalue), log\_usage(v)}\end{array}} \quad (6)$$

(c) Assignment Statement

$$\frac{\begin{array}{l}\texttt{is\_traced(v), in\_callee(v),}\\ \texttt{is\_simple(v), flow\_to(v,callee\_arg)}\end{array}}{\begin{array}{l}\texttt{set\_alias(callee\_arg,get\_alias(v)),}\\ \texttt{set\_is\_traced(callee\_arg), log\_usage(v)}\end{array}} \quad (7)$$

$$\frac{\begin{array}{l}\texttt{is\_traced(v), in\_callee(v),}\\ \texttt{in\_simple\_expr(v), flow\_to(v,callee\_arg)}\end{array}}{\begin{array}{l}\texttt{set\_alias(callee\_arg,refine\_expr(get\_alias(v))),}\\ \texttt{set\_is\_traced(callee\_arg), log\_usage(v)}\end{array}} \quad (8)$$

(d) Callee Statement

$$\frac{\begin{array}{l}\texttt{is\_traced(v), flow\_to(v,caller\_arg),}\\ \texttt{is\_simple(caller\_arg)}\end{array}}{\begin{array}{l}\texttt{set\_alias(caller\_arg,get\_alias(v)),}\\ \texttt{set\_is\_traced(caller\_arg), log\_usage(v)}\end{array}} \quad (9)$$

(e) Caller Statement

```
is_simple(x): var |ptr |ref |idx |memptr |memref
e.g., (x, *x, &x, x[y], x.m, x->m)
in_simple_expr(x): add |sub
e.g., (x-y, x+y)
```

(f) Notation Explains

Figure 7: Inference rules used in data flow analysis for ReCopilot.

Throughout the analysis process, we maintain and update alias relationships between the current variables and their original traced variables, indicating how each traced variable is used at different locations. For example, Figure 6 shows a prompt for argument recovery task on `sub_1903@a1`, and we traced `a1` through the call graph to track its propagation and usages. We finally annotated the alias relationship at the variable usage location, *e.g.*, the `__int64 a2` in `sub_14F7` is a direct alias of the original `a1` in the target function. With these alias annotations, the model is directly aware of the usages of the target variable at all locations that are referring to it without needing to perform analysis itself. Our lightweight analysis yields impressive efficiency, specifically tracking all variables for each function takes 0.0182s and 0.0578s for one- and two-level context, respectively.

4 Benchmark

As discussed in §2, existing domain benchmarks in binary analysis typically focus on only one or two specific tasks. To comprehensively evaluate our ReCopilot and the baselines, we constructed a multi-task benchmark as shown in Figure 8. Our goal was to create an automatic and extensible evaluation pipeline for the LLM-based reverse engineering tools. Given that the most popular baselines (detailed in §5) are integrated with IDA Pro [1] as plugins, we employed it to build the runtime environment first, which can be easily extended to other decompilation platforms in the future. This pipeline takes a binary as input and triggers a plugin (i.e., a baseline method) to analyze the targeted function at once. File-level inputs, rather than function-level used by other benchmarks, provide sufficient context to the potential usage by the evaluation objects. Further, the analysis results are persisted by IDA Pro into a `.idb` file, from which our extractor parses the predictions to run the evaluation.

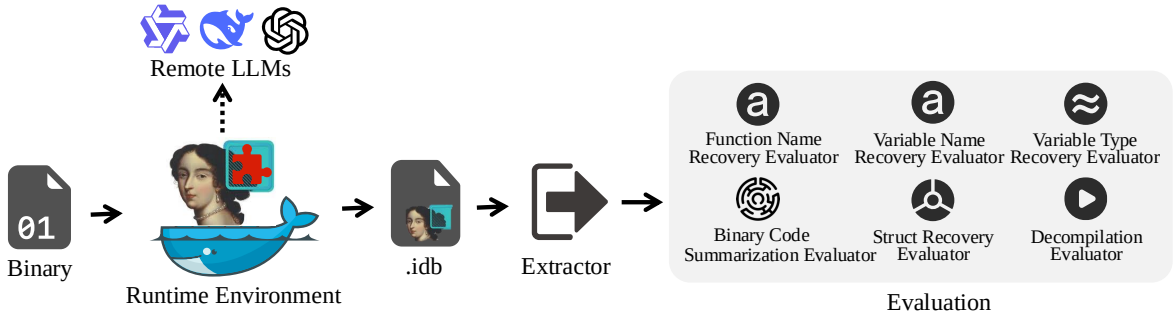

Figure 8: An overview of our benchmark for evaluating binary analysis tools.

Our benchmark covers the most important meta-tasks identified from all analysis tasks supported by related works, and we designed corresponding evaluators to provide accurate measurement and valuable insights. These tasks and evaluators are detailed as follows:

- **Function Name Recovery.** We evaluate this task with the Rouge score [68], which is a recall-oriented metric to indicate how many ground-truth tokens are recalled into the prediction.
- **Variable Name Recovery.** Since the variable name is similar to the function name, we reuse the same metric.
- **Variable Type Recovery.** In this task, we evaluate only the basic types (*e.g.*, int, float, and char), which appear in different forms in the pseudo code decompiled by IDA Pro. We cluster these equivalent types together, for instance, both `__int64` and `unsigned __int64` represent the same type sense with identical memory size. A prediction is considered correct if it belongs to the same cluster as the ground truth. In addition, we ignore the type qualifiers such as `const`.
- **Struct Recovery.** Our primary focus here is on recognizing memory patterns of structures, particularly the identification of member numbers and sizes. In practice, understanding memory layouts of complex variables plays a crucial role in enabling further program analysis, such as taint tracking. We evaluate each prediction for this task by computing precision and recall of predicted member boundaries, and then derive the F1 score as the final metric:

$$\text{Precision} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FP}|}, \quad \text{Recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}, \quad \text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{10}$$

  where TP represents correctly predicted member boundaries, FP denotes incorrectly predicted boundaries, and FN indicates ground-truth boundaries that were not predicted.
- **Binary Code Summarization.** Since LLM-as-a-judge had been demonstrated to have a strong alignment with human preference [69], with reference to the empirical study [70] and the real-world practice [71], we employ

a LLM to evaluate generated summarization across the following key dimensions: ① Semantic Coverage, ② Semantic Accuracy, ③ Misleading, and ④ Readability. For each dimension, the LLM makes a binary judgment (yes/no), and the final score is computed as the equally weighted average of these four dimensions, resulting in an overall score between 0 and 1. For example, if a summarization receives positive judgments in three dimensions but fails in one, it would receive a score of 0.75.

- **Decompilation.** To evaluate the quality of decompiled code, we employ CodeBLEU [72], a comprehensive metric widely adopted in code synthesis and translation tasks. The CodeBLEU score is computed by comparing the tool-generated decompiled code against the original source code across three aspects: syntactic structure (AST), data flow dependencies (DFG), and lexical matching (n-grams).

Test Dataset. The test dataset is built to cover as many domains as possible, specifically, we selected open-source repositories from crypto, network, multimedia, compression, database, sys-utils, etc., and we compiled them with Linux and Windows runtimes. Similar to the data collection in §3.3.1, we employed sanitizer to filter out noise and prevent data leakage from our training dataset. Although we have no access to the training data of the baseline models, our test dataset is constructed from private environments, suggesting that our data are unlikely to be encountered by them during their training. In total, we sampled 1,038 binary functions as targets from 60 binaries, which were compiled from 16 projects.

General Benchmarks. To further evaluate the general capabilities of our model, we employed a suite of widely recognized benchmarks that target distinct aspects of LLM performance:

- **Mathematical Reasoning:** We use MATH [73] to evaluate model's proficiency in handling complex mathematical concepts and multi-step reasoning processes.
- **Knowledge and Reasoning:** MMLU [74] is used to assess the model's knowledge across a wide range of domains, while GPQA-Diamond [75] specifically assesses the capabilities for expert-level reasoning and understanding in complex questions.
- **Code Generation:** To evaluate the programming abilities across diverse tasks and languages, we utilized HumanEval [16] and MBPP [76]. These benchmarks focus on the model's capability to generate accurate and functional source code.
- **Instruction Following:** IFEval [77] is used to measure the model's ability to understand and follow diverse instructions.

These additional benchmarks provide a multidimensional view of the model's general capabilities, allowing us to analyze how domain-specific training influences performance across mathematical reasoning, broad knowledge application, code synthesis, and instruction comprehension. We leveraged the OpenCompass [71] evaluation platform to systematically benchmark our model alongside baseline LLMs on these tasks.

5 Evaluation

5.1 Experiment Settings

Training Settings. We conducted all experiments on two Linux servers, each equipped with 1TB RAM and 8 * NVIDIA A800-80GB GPUs. We used `Qwen2.5-Coder-7B` [56] as the base model and applied our training strategy to produce the ReCopilot model. The model checkpoints from each training stage are denoted as `recopilot-v0.1-beta-*`, with the suffix indicating the stage. For example, `recopilot-v0.1-beta-dpo` is our final model trained by DPO. During training, the maximum context length of our model is set to 32K tokens to ensure the effective learning of super-long CoT data. We acknowledge the support of open-source training frameworks such as Transformers [78], TRL [79], LLaMA-Factory [80], and DeepSpeed [81], which enable us to conduct efficient training. Finally, a full model training goes through CPT, SFT, and DPO stages, and takes about 7.1K GPU hours in total.

Evaluation Settings. For ReCopilot, by default, we configure the system with a maximum trace depth of 1 for both callee and caller, a maximum of 10 context functions, and a maximum output length of 16K tokens. All of the LLM-based methods have inherent randomness, we thus set a maximum of 3 retries for the non-applicable prediction, such as format errors. Our model uses the prompt template shown in Figure 4, which supports all evaluated tasks by specifying task tags, labeled as `recopilot`. We evaluate baseline methods with the default prompt designed by themselves. Our benchmark employed IDA Pro 9.0 as the infrastructure platform in the runtime environment.

Baselines Selection. We investigated the existing LLM-based tools and methods [82, 83, 84, 10, 85, 86, 11, 35] for the analysis tasks we focused on. The following ones are widely known in the reverse-engineering community, we thus selected them as our baselines:

- Driven by General LLM: The Gepetto [82] and WPeChatGPT [83] are two open-source tools designed for harnessing general LLMs to analyze binary code, supporting variable name recovery and binary code summarization tasks.
- Driven by Tailored LLM: The aidapal [84] and LLM4Decompile [10] build their expert LLM for specific tasks. The aidapal supports function name recovery, variable name recovery, and binary code summarization, while the LLM4Decompile supports the decompilation task only. We also include a commercial tool, BinaryNinja [87], that provides an AI-powered official extension, *i.e.*, Sidekick. We evaluate it on function name recovery, variable name recovery, and summarization.

Beyond these tools, we select state-of-the-art LLMs as the model baselines for model comparison, considering that our ReCopilot can also be driven by a general-purpose LLM with a tailored prompt. The `Qwen2.5-Coder-7B-Instruct` [56] is derived from the same base model as ours, and it demonstrates top-tier performance among models of the 7B parameter size. We further take DeepSeek-V3 (671B) [38] and DeepSeek-R1 [41] as representative examples of advanced large-scale LLMs and reasoning LLMs, respectively, and compare the performance of our model with theirs.

5.2 Overall Effectiveness

Table 4: An overall results of comparison against existing tools on binary analysis tasks.

Tool	Model	Prompt[2]	Succ	Func Name	Var Name	Var Type	Struct	Dec	Sum
Gepetto	DeepSeek-V3(671B)	Self-Def	**1.00**	/	7.59	/	/	/	58.26
WPeChatGPT	DeepSeek-V3(671B)	Self-Def	**1.00**	/	7.79	/	/	/	65.15
aidapal	aidapal-8k(7B)	Self-Def	0.99	9.99	9.9	/	/	/	61.21
LLM4Decompile	LLM4Decompile-9B-v2	Self-Def	**1.00**	/	/	/	/	25.51	/
BinaryNinja[3]	Sidekick 3.0	/	0.85	34.55	23.76	/	/	/	60.85
recopilot-v0.1-beta	recopilot-v0.1-beta-dpo(7B)[1]	recopilot	0.92	**50.59**	**43.50**	**39.81**	**27.67**	**29.23**	**65.21**

[1] `recopilot-v0.1-beta-dpo` is the checkpoint of our expert LLM after DPO training, *i.e.*, the final model.
[2] Self-Def represents self-defined prompts, while recopilot is the prompt template used by our model.
[3] BinaryNinja is a commercial tool that we have no idea about its internal details, and we evaluate it through running the Sidekick extension in batch.

Using our binary analysis benchmark, we conducted experiments to evaluate the performance of our ReCopilot and the baselines in the domain tasks. We deployed DeepSeek-V3 to power tools designed with general-purpose LLMs, while the tools initiated by tailored models employed their own models. For ReCopilot, we used the `recopilot-v0.1-beta-dpo` model with the `recopilot` prompt and took the configuration by default detailed above.

As shown in Table 4, the results demonstrate that ReCopilot significantly outperforms all baseline methods on almost all tasks, indicating a solid overall effectiveness. In particular, ReCopilot achieves an average outperformance of 13% over the 2nd place across all tasks. Also, our method implements the functions of variable type recovery and struct recovery that were unsupported by previous LLM-based tools. Although ReCopilot currently fulfills the goal of serving as a human assistant, our evaluation results indicate that its absolute performance still has room for improvement before it can be reliably applied to downstream tasks that demand high-level soundness.

As mentioned earlier, the baselines essentially consider a few tasks only, resulting in many blanks in their assessment results (Table 4), which also suggests their limited value for practical deployment. In contrast, our ReCopilot model has acquired instruction-following capabilities through the training on both general-purpose and domain-specific tasks, enabling it to perform arbitrary tasks following user prompts. This adaptability broadens its applicability across a wider range of scenarios and enhances its practical significance.

For the particular tasks, we have observed that ReCopilot achieved the most significant advantage of performance on the variable name recovery task, surpassing the previous best method BinaryNinja by 19.74%. Compared to LLM4Decompile, which is specifically designed for the decompilation task, our method still exhibits a 3.71% performance advantage. Regarding binary code summarization, an interesting fact is that the general-purpose LLM also generates summaries well. Specifically, WPeChatGPT and Gepetto with DeepSeek-V3 respectively scored 58.26% and 65.15% on this task, which are close to the 65.21% score of our expert model. It suggests that even non-expert LLMs have shown promising results in assisting binary understanding.

We also count the ratio of LLM generation being successfully applied, denoted as success ratio, presented in the "Succ" column of Table 4. The results indicate that ReCopilot struggles to generate format-correct and syntactically

correct predictions, reaching a 92% success rate lower than the other methods. Notably, the other methods require only format-agnostic and syntax-agnostic generation, *i.e.*, the model raw output is directly inserted into the placeholders such as comments or function names. However, our model must organize predictions into JSON format, and these predictions may contain errors that make them unusable, such as predicting non-existent variable types. This weakness indicates a potential direction for ReCopilot improvement.

5.3 Model Comparison

Table 5: Comparison of ReCopilot performance with different models on binary analysis tasks.

Model	Prompt	Succ	Func Name	Var Name	Var Type	Struct	Dec	Sum
Qwen2.5-Coder-7B-Instruct	general	0.71	34.57	24.07	30.27	9.09	23.13	50.49
DeepSeek-V3	general	0.85	40.20	18.21	34.75	14.47	25.20	56.90
DeepSeek-R1	general_wo_guide	0.84	43.63	25.36	38.92	22.12	**29.77**	**66.32**
recopilot-v0.1-beta-Qwen-dpo[1]	recopilot	**0.92**	46.66	32.98	37.91	26.45	26.19	62.15
recopilot-v0.1-beta-dpo	recopilot	**0.92**	**50.59**	**43.50**	**39.81**	**27.67**	29.23	65.21

[1] `recopilot-v0.1-beta-Qwen-dpo` is a final checkpoint trained without the CPT stage, deriving from the same base model with `recopilot-v0.1-beta-dpo`.

We further conducted experiments to compare the model performance. The different LLMs are used to drive our ReCopilot tool with the same context enhancement applied, using the default settings. Meanwhile, we designed two prompts, `general` and `general_wo_guide`, to instruct the non-expert models to generate final results in specific JSON format. These prompts contain the role description, task statement, formatting instruction, and share the same input specification with the `recopilot` prompt (shown in Figure 4). The `general` prompt includes the expert-written stepwise guide for the current analysis task, which is also used in building our SFT dataset (§3.3.3), whereas the guide is removed from the `general_wo_guide` prompt. We use the `general` prompt with regular LLMs (*e.g.*, DeepSeek-V3) for better performance, as these models usually lack domain skills. Since the reasoning LLMs (*e.g.*, DeepSeek-R1) have already developed their own reasoning habits, we use another prompt without any guide to set their thoughts free.

The experimental results are shown in Table 5. Among all LLMs, our model, `recopilot-v0.1-beta-dpo`, achieved the best performance across most tasks, especially showed significant dominance in variable name recovery task, leading the 2nd place DeepSeek-R1 by a 71% relative score. According to publicly available leaderboards [88, 71], `Qwen2.5-Coder-7B-Instruct` is a leading model in the 7B parameter scale. The average score of our model across all tasks exceeds it by 14%, indicating a significant performance advantage for our expert model. Even when compared to advanced large-scale reasoning LLM, DeepSeek-R1, our model still has a small advantage of 5%. In addition, the smaller size of the ReCopilot model implies lower resource requirements, overcoming the challenge of laptop-deployability we proposed in §2. Furthermore, we directly conducted post-training to the base model without domain pretraining, and produced the `recopilot-v0.1-beta-Qwen-dpo` model. It only achieved sub-optimal performance in the binary analysis tasks compared to the checkpoint with full training performed, indicating that the model can effectively learn domain knowledge through CPT.

Table 6: Comparison of LLM performance on the benchmarks in the general domains.

Models	MATH	MMLU	IFEval	HumanEval	MBPP	GPQA-diamond
DeepSeek-V3	**81.70**	**86.71**	**82.09**	**91.10**	**81.67**	**56.11**
Qwen2.5-Coder-7b-Instruct	43.30	65.52	58.04	84.76	73.80	31.31
recopilot-v0.1-beta-Qwen-dpo	37.14	63.97	57.67	82.32	69.40	34.34
recopilot-v0.1-beta-dpo	33.34	64.34	56.93	78.05	61.60	25.76

[*] The scores are practically obtained through our evaluation environment, and minor differences compared to their original reports do not invalidate our comparison of their relative performance.

Beyond our domain benchmark, we also employ widely recognized benchmarks to evaluate general capabilities. It is important to note that our primary interest does not lie in the evaluation of the absolute performance. Rather, we aim to provide additional insights by examining how domain-specific training influences the model's general capabilities through comprehensive assessment. We utilize `Qwen2.5-Coder-7b-Instruct` as a reference baseline to illustrate the impact of our training methodology.

As demonstrated in Table 6, both of the two ReCopilot models generally underperform compared to the baseline model on general-purpose benchmarks. Specifically, `recopilot-v0.1-beta-dpo`, which went through all three

training stages, exhibited the lowest performance, averaging 6.12% below the baseline across all tasks. This result suggests that our domain-specific training has indeed compromised the model performance in the general domains. However, the `recopilot-v0.1-beta-Qwen-dpo` model, which went through only the SFT and DPO stages, experienced significantly less damage, with an average performance decline of just 1.98%. Notably, we also observe that `recopilot-v0.1-beta-Qwen-dpo` outperformed the baseline by 3.03% in the complex reasoning assessment (*i.e.*, GQPA-diamond). This improvement may be attributed to the inclusion of a large amount of general and domain-specific reasoning data in the SFT dataset. In addition, `recopilot-v0.1-beta-dpo` consistently underperforms compared to `recopilot-v0.1-beta-Qwen-dpo` across most benchmarks, with an average performance gap of 4.13%. This disparity suggests that the CPT training stage may have led to a greater degree of "knowledge forgetting", resulting in the model losing more world knowledge and general capabilities.

5.4 Ablation Study

Table 7: Results of ablation experiments on Super-CoT, task guide, DPO training, and context enhancement.

Model	Prompt	Succ	Func Name	Var Name	Var Type	Struct	Dec	Sum
DeepSeek-V3-wo-DFA	general	0.87	35.11	17.01	34.17	9.77	24.99	59.78
DeepSeek-V3	general	0.85	40.20	18.21	34.75	14.47	25.20	56.90
DeepSeek-V3	general_wo_guide	**0.88**	35.61	18.76	30.77	14.42	24.75	60.22
DeepSeek-R1	general	0.79	**46.61**	24.87	37.76	11.38	**30.84**	64.98
DeepSeek-R1	general_wo_guide	0.84	43.63	**25.36**	**38.92**	**22.12**	29.77	**66.32**
recopilot-v0.1-beta-sft	recopilot	0.82	**51.17**	**44.01**	**40.89**	**31.55**	**30.11**	**67.60**
recopilot-v0.1-beta-dpo	recopilot	**0.92**	50.59	43.50	39.81	27.67	29.23	65.21
recopilot-v0.1-beta-dpo	recopilot_super_thought	**0.92**	50.75	43.52	39.03	29.23	28.87	63.91

Super-CoT. We introduced the concept of super long chain-of-thought (Super-CoT) in §3.3.3, and used a new thinking tag (`<Super-Thought>`) to prompt our model to engage in deep and extended reasoning. This prompt is labeled as `recopilot_super_thought` here. We first conducted an ablation study to evaluate the impact of Super-CoT on the model's performance. As shown in Table 7, the performance metrics for `recopilot-v0.1-beta-dpo` with Super-CoT are similar to the one without it. One potential reason for this limited effectiveness is the dataset imbalance. Our Super-CoT dataset comprises only 11K examples, whereas the standard reasoning SFT dataset contains 100K examples. This disparity in dataset size may have hindered the model to fully leverage the benefits of deep reasoning, suggesting that expanding the Super-CoT dataset could be a potential direction for our future efforts. Additionally, we have successfully leveraged SFT to equip the model with basic reasoning capabilities, but it seems to struggle to achieve stable deep reasoning. Given the previous studies [41, 89], reinforcement learning is another potential way for improvement.

Task Guide for General LLM. We further conducted an evaluation to investigate the `general` prompts, revealing that general LLMs like DeepSeek-V3 benefit from task-specific guides. With guided prompts, DeepSeek-V3 achieved a higher average score of 31.62%, compared to 30.60% without guidance, indicating that guided prompts help the general-purpose model skilled binary analysis tasks. Conversely, for reasoning models like DeepSeek-R1, removing the guide improved performance, with an average score increase from 36.01% to 37.68%. This indicates that such models perform better when allowed to conduct their inherent reasoning capabilities without constraints.

DPO. We evaluated the `recopilot-v0.1-beta-sft` model, a prior checkpoint of `recopilot-v0.1-beta-dpo`, which had not yet undergone DPO training. By comparing the evaluation results of these two models, we observed that DPO training significantly enhances the success ratio, increasing it from 0.82 to 0.92. This improvement underscores the effectiveness of DPO training in refining the model's ability to produce format-correct and syntactically accurate predictions, which are essential for the integration into decompiled code. While the success rate has improved, there are declines in certain tasks, such as struct recovery (3.87%) and decompilation (0.87%). This suggests that while DPO optimizes the model's prediction reliability, it may slightly compromise its performance in specific tasks. These findings highlight the importance of balancing the preference alignment and the original performance.

Data Flow Analysis. In order to evaluate the impact of data flow analysis (DFA), as shown in Table 7, we conduct a comparison and present results in the rows of `DeepSeek-V3` and `DeepSeek-V3-wo-DFA`. Our ReCopilot model is trained with data that includes DFA, performing ablation on them would naturally lead to performance degradation due to missing information. Therefore, we chose to conduct this ablation experiment on the DeepSeek-V3 model to isolate and assess the specific contribution of DFA to the analysis performance of LLMs. The evaluation results demonstrate that the inclusion of DFA significantly improved performance in specific tasks. For instance, DeepSeek-V3 with DFA exhibits 4.70% higher accuracy in identifying struct layouts, highlighting the importance of understanding variable

propagation and usage patterns. This improvement underscores the value of DFA in providing the model with a more comprehensive understanding of the code's data flow, thereby facilitating more accurate and insightful analysis.

6 Discussion

In the previous section, the evaluation demonstrates that our method outperforms the baseline methods. Although the absolute performance of ReCopilot still needs to be further improved, it has shown the potential of becoming the next generation of decompilation assistants. This section will discuss the limitations of our work and potential ways for future improvements. Starting with the weaknesses of ReCopilot in practice:

Weakness on Supported Tasks. Binary analysis is a complex engineering that involves different analytical tasks depending on the objectives. For instance, malware analysis often requires code clustering to identify functional modules, while deobfuscation may necessitate control flow optimization. Constrained by computation resource, ReCopilot currently focuses primarily on the most fundamental and critical tasks for decompilation augmentation, which leads us to temporarily overlook higher-level and peripheral requirements.

Weakness on Supported Binary Representations. ReCopilot currently works on decompiled pseudo code only, which leads to weaknesses in handling other representations of binary code, such as disassembly code. In practice, some binaries for specialized CPU architectures lack a decompiler to obtain pseudo code. We plan to extend our model's ability in tackling disassembly code in the future, thereby supporting more practical scenarios.

Weakness on Supported Programming Languages. Recently, a growing number of compiled high-level languages have been widely used, such as Go and Rust. However, our training data are exclusively coming from C/C++ projects, which prevents our model from learning binaries built using other programming languages. The baseline methods suffer from the same problem, underscoring a pressing need for dataset expansion.

The weaknesses presented above could be mitigated by taking more efforts on dataset building and model training on our current method, that is, there are no serious obstacles. Furthermore, we identified several potential approaches to improve the performance of our model in the future:

Reinforcement Learning. Reinforcement learning in the LLM domain has shown significant efficacy in improving model performance [43, 89, 41]. While our model has acquired a certain level of reasoning ability through SFT directly, it struggles to perform stable and consistent logical reasoning in a super long CoT, detailed in the ablation study (§5.4). However, the on-policy reinforcement learning, which continuously optimizes the model starting from itself, could help the model naturally develop robust ability of test-time scaling.

Model and Dataset Scaling. The neural scaling law [90] has been empirically validated, stating that model performance improves as both model size and dataset size increase. Our current 7B model is designed to be deployable on personal laptop, which inherently limits its upper bound. A promising direction for optimization is to scale up the model parameters, as well as the diversity of data and the number of domain tasks.

Agentic Mode. LLM-driven agents have achieved promising results in automating sophisticated tasks, as exemplified by tools like Deep Research [91] and Cursor [92]. Unlike traditional LLMs, which primarily engage in Q&A interactions, agents are capable of autonomous planning to solve problems through multiple steps and invoke external tools to obtain additional information or assistance. Building agentic capabilities on top of ReCopilot model presents a promising approach to addressing more complex binary analysis challenges.

7 Conclusion

In this work, we presented ReCopilot, an expert LLM, tailored to provide assistance with reverse engineers in binary analysis. To build the model, we took efforts to collect large-scale raw dataset and devised a generator-discriminator framework to construct CoT data. We further employed context enhancement through data flow and call graph analysis for better performance. A benchmark has been implemented for binary analysis with the most important tasks supported. Our comprehensive evaluation showed that ReCopilot outperforms existing domain-specific LLMs and advanced general LLMs. We have elaborated on the implementation details of ReCopilot and demonstrated this work to the security community. We hope this work promotes the security community to drive binary reverse engineering into the next generation.

References

[1] Hex-RaysSA. "ida pro". `https://www.hex-rays.com/products/ida`, 2025.

[2] NationalSecurityAgency. "ghidra". https://github.com/NationalSecurityAgency/ghidra, 2025.

[3] Mira Leung and Gail Murphy. On automated assistants for software development: The role of llms. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1737–1741. IEEE, 2023.

[4] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[5] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. URL: https://arxiv.org/abs/2406.11931, arXiv:2406.11931.

[6] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE, 2023.

[7] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. How far have we gone in vulnerability detection using large language models. *arXiv preprint arXiv:2311.12420*, 2023.

[8] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2312–2323. IEEE, 2023.

[9] OpenAI, :, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Mürk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. Competitive programming with large reasoning models, 2025. URL: https://arxiv.org/abs/2502.06807, arXiv:2502.06807.

[10] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487, Miami, Florida, USA, November 2024. Association for Computational Linguistics. URL: https://aclanthology.org/2024.emnlp-main.203/, doi:10.18653/v1/2024.emnlp-main.203.

[11] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[12] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4069–4087, 2024. doi:10.1109/SP54263.2024.00152.

[13] kokke, 2025. URL: https://github.com/kokke/tiny-AES-c.

[14] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[15] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*, 2024. URL: https://openreview.net/forum?id=IBCBMeAhmC.

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

[17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL: https://arxiv.org/abs/2310.06770, arXiv:2310.06770.

[18] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601*, 2023.

[19] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. How far have we gone in stripped binary code understanding using large language models. *arXiv e-prints*, pages arXiv–2404, 2024.

[20] Dylan Manuel, Nafis Tanveer Islam, Joseph Khoury, Ana Nunez, Elias Bou-Harb, and Peyman Najafirad. Enhancing reverse engineering: Investigating and benchmarking large language models for vulnerability analysis in decompiled binaries, 2024. URL: `https://arxiv.org/abs/2411.04981`, `arXiv:2411.04981`.

[21] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1667–1680, 2018. `doi:10.1145/3243734.3243866`.

[22] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. `doi:10.1145/3428293`.

[23] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 607–619, 2021.

[24] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022.

[25] Guoqiang Chen, Han Gao, Jie Zhang, Yanru He, Shaoyin Cheng, and Weiming Zhang. Investigating neural-based function name reassignment from the perspective of binary code representation. In *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*, pages 1–11. IEEE, 2023.

[26] James Patrick-Evans, Moritz Dannehl, and Johannes Kinder. Xfl: Naming functions in binaries with extreme multi-label learning. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2375–2390. IEEE, 2023.

[27] Xiaoling Zhang, Zhengzi Xu, Shouguo Yang, Zhi Li, Zhiqiang Shi, and Limin Sun. Enhancing function name prediction using votes-based name tokenization and multi-task learning. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. `doi:10.1145/3660782`.

[28] Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin`.

[29] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. Direct: A transformer-based model for decompiled identifier renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 48–57, 2021.

[30] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639, Nov 2019. `doi:10.1109/ASE.2019.00064`.

[31] Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. Hext5: Unified pre-training for stripped binary code information inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 774–786. IEEE, 2023.

[32] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 690–702, 2021.

[33] Ziyi Lin, Jinku Li, Bowen Li, Haoyu Ma, Debin Gao, and Jianfeng Ma. Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2725–2739, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3576915.3623214`.

[34] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. TYGR: Type Inference on Stripped Binaries using Graph Neural Networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, 2024.

[35] Zirui Song, Yutong Zhou, Shuaike Dong, Ke Zhang, and Kehuan Zhang. Typefsl: Type prediction from binaries via inter-procedural data-flow analysis and few-shot learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1269–1281, 2024.

[36] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binarie. In *2023 IEEE*

International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 260–271. IEEE, 2023.

[37] Aaron Grattafiori, Abhimanyu Dubey, and etc. Abhinav Jauhri. The llama 3 herd of models, 2024. URL: https://arxiv.org/abs/2407.21783, arXiv:2407.21783.

[38] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, and etc. Deepseek-v3 technical report, 2025. URL: https://arxiv.org/abs/2412.19437, arXiv:2412.19437.

[39] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL: https://arxiv.org/abs/2412.15115, arXiv:2412.15115.

[40] OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, and Ahmed El-Kishky etc. Openai o1 system card, 2024. URL: https://arxiv.org/abs/2412.16720, arXiv:2412.16720.

[41] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, and etc. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL: https://arxiv.org/abs/2501.12948, arXiv:2501.12948.

[42] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL: https://arxiv.org/abs/2305.18290, arXiv:2305.18290.

[43] 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. Training language models to follow instructions with human feedback, 2022. URL: https://arxiv.org/abs/2203.02155, arXiv:2203.02155.

[44] archlinux.org. Arch linux packages, 2025. URL: https://archlinux.org/packages/.

[45] ubuntu.com. Ubuntu packages, 2025. URL: https://packages.ubuntu.com/.

[46] debian.org. Debian packages, 2025. URL: https://www.debian.org/distrib/packages.

[47] Anonymous. Compileagent: Automated real-world repo-level compilation with tool-integrated LLM-based agent system. In *Submitted to ACL Rolling Review - December 2024*, 2025. under review. URL: https://openreview.net/forum?id=1kic2XYZiR.

[48] tree sitter. "tree-sitter". https://github.com/tree-sitter/tree-sitter, 2025.

[49] A.Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997. doi:10.1109/SEQUEN.1997.666900.

[50] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024. URL: https://arxiv.org/abs/2402.19173, arXiv:2402.19173.

[51] Maurice Weber, Daniel Y. Fu, Quentin Anthony, Yonatan Oren, Shane Adams, Anton Alexandrov, Xiaozhong Lyu, Huu Nguyen, Xiaozhe Yao, Virginia Adams, Ben Athiwaratkun, Rahul Chalamala, Kezhen Chen, Max Ryabinin, Tri Dao, Percy Liang, Christopher Ré, Irina Rish, and Ce Zhang. Redpajama: an open dataset for training large language models. *NeurIPS Datasets and Benchmarks Track*, 2024.

[52] Wikimedia Foundation. Wikimedia downloads, 2025. URL: https://dumps.wikimedia.org.

[53] mikex86. stackoverflow-posts, 2025. URL: https://huggingface.co/datasets/mikex86/stackoverflow-posts.

[54] clouditera. security-paper-datasets, 2025. URL: https://huggingface.co/datasets/clouditera/security-paper-datasets.

[55] Jiawei Gu, Zacc Yang, Chuanghao Ding, Rui Zhao, and Fei Tan. CMR scaling law: Predicting critical mixture ratios for continual pre-training of language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16143–16162, Miami, Florida, USA, November 2024. Association for Computational Linguistics. URL: https://aclanthology.org/2024.emnlp-main.903/, doi:10.18653/v1/2024.emnlp-main.903.

[56] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL: https://arxiv.org/abs/2409.12186, arXiv:2409.12186.

[57] Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas, 2024. URL: https://arxiv.org/abs/2406.20094, arXiv:2406.20094.

[58] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL: https://arxiv.org/abs/2411.15124, arXiv:2411.15124.

[59] Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL: https://openreview.net/forum?id=1Iuw1jcIrf.

[60] Qinyuan Ye. Cross-task generalization abilities of large language models. In Yang (Trista) Cao, Isabel Papadimitriou, Anaelia Ovalle, Marcos Zampieri, Francis Ferraro, and Swabha Swayamdipta, editors, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*, pages 255–262, Mexico City, Mexico, June 2024. Association for Computational Linguistics. URL: https://aclanthology.org/2024.naacl-srw.27/, doi:10.18653/v1/2024.naacl-srw.27.

[61] Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL: https://huggingface.co/datasets/teknium/OpenHermes-2.5.

[62] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024. URL: https://openreview.net/forum?id=CfXh93NDgH.

[63] Shubham Toshniwal, Wei Du, Ivan Moshkov, Branislav Kisacanin, Alexan Ayrapetyan, and Igor Gitman. Openmathinstruct-2: Accelerating ai for math with massive open-source instruction data. *arXiv preprint arXiv:2410.01560*, 2024.

[64] Open-Source-O1. Open-O1. https://huggingface.co/datasets/O1-OPEN/OpenO1-SFT, 2024.

[65] OpenThoughts Team. Open Thoughts. https://open-thoughts.ai, January 2025.

[66] joern.io. Joern: The Bug Hunter's Workbench, January 2024. URL: https://github.com/joernio/joern.

[67] semgrep. Semgrep. https://github.com/semgrep/semgrep, 2025.

[68] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, 2004. Association for Computational Linguistics. URL: https://aclanthology.org/W04-1013.

[69] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

[70] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. A survey on llm-as-a-judge, 2025. URL: https://arxiv.org/abs/2411.15594, arXiv:2411.15594.

[71] OpenCompass Contributors. Opencompass: A universal evaluation platform for foundation models. https://github.com/open-compass/opencompass, 2023.

[72] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[73] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023.

[74] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021. URL: https://arxiv.org/abs/2009.03300, arXiv:2009.03300.

[75] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL: https://arxiv.org/abs/2311.12022, arXiv:2311.12022.

[76] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL: https://arxiv.org/abs/2108.07732, arXiv:2108.07732.

[77] Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models, 2023. URL: https://arxiv.org/abs/2311.07911, arXiv:2311.07911.

[78] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. pages 38–45. Association for Computational Linguistics, October 2020. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[79] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformer Reinforcement Learning. URL: https://github.com/huggingface/trl.

[80] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In Yixin Cao, Yang Feng, and Deyi Xiong, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, Bangkok, Thailand, August 2024. Association for Computational Linguistics. URL: https://aclanthology.org/2024.acl-demos.38/, doi:10.18653/v1/2024.acl-demos.38.

[81] DeepSpeed, 2025. URL: https://www.deepspeed.ai/.

[82] JusticeRage. Gepetto. https://github.com/JusticeRage/Gepetto, 2022.

[83] WPeace-HcH. Wpechatgpt. https://github.com/WPeace-HcH/WPeChatGPT, 2023.

[84] atredispartners. aidapal. https://github.com/atredispartners/aidapal, 2024.

[85] Mlm. https://mlm01.com, 2024.

[86] mrphrazer. reverser_ai. https://github.com/mrphrazer/reverser_ai, 2024.

[87] Vector35. "binary ninja". https://binary.ninja/, 2025.

[88] EvalPlus, 2024. URL: https://evalplus.github.io/leaderboard.html.

[89] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL: https://arxiv.org/abs/2402.03300, arXiv:2402.03300.

[90] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL: https://arxiv.org/abs/2001.08361, arXiv:2001.08361.

[91] OpenAI. Introducing deep research. https://openai.com/index/introducing-deep-research/, 2025.

[92] Anysphere Inc. TRL: Transformer Reinforcement Learning, 2025. URL: https://www.cursor.com.