# Source Code Foundation Models are Transferable Binary Analysis Knowledge Bases

**Zian Su**[*]
Purdue University
su284@purdue.edu

**Xiangzhe Xu**
Purdue University
xu1415@purdue.edu

**Ziyang Huang**
Purdue University
huan1562@purdue.edu

**Kaiyuan Zhang**
Purdue University
zhan4057@purdue.edu

**Xiangyu Zhang**
Purdue University
xyzhang@cs.purdue.edu

## Abstract

Human-Oriented Binary Reverse Engineering (HOBRE) lies at the intersection of binary and source code, aiming to lift binary code to human-readable content relevant to source code, thereby bridging the binary-source semantic gap. Recent advancements in uni-modal code model pre-training, particularly in generative Source Code Foundation Models (SCFMs) and binary understanding models, have laid the groundwork for transfer learning applicable to HOBRE. However, existing approaches for HOBRE rely heavily on uni-modal models like SCFMs for supervised fine-tuning or general LLMs for prompting, resulting in sub-optimal performance. Inspired by recent progress in large multi-modal models, we propose that it is possible to harness the strengths of uni-modal code models from both sides to bridge the semantic gap effectively. In this paper, we introduce a novel probe-and-recover framework that incorporates a binary-source encoder-decoder model and black-box LLMs for binary analysis. Our approach leverages the pre-trained knowledge within SCFMs to synthesize relevant, symbol-rich code fragments as context. This additional context enables black-box LLMs to enhance recovery accuracy. We demonstrate significant improvements in zero-shot binary summarization and binary function name recovery, with a 10.3% relative gain in CHRF and a 16.7% relative gain in a GPT4-based metric for summarization, as well as a 6.7% and 7.4% absolute increase in token-level precision and recall for name recovery, respectively. These results highlight the effectiveness of our approach in automating and improving binary code analysis.

## 1 Introduction

In recent years, we see two trends of uni-modal code model pre-training. On one hand, there is a remarkable surge in the development of generative Source Code Foundation Models (SCFMs) [12, 64, 55, 22, 43], along with advancements in general Large Language Models (LLMs) [4, 59, 46]. Driven by a growing interest in automating software development, these powerful models are trained on billions of tokens from diverse codebases, covering a wide spectrum of programming languages [35]. They possess the capability to complete, infill [20], and refine code [66], as well as generate code from natural language instructions [44]. On the other hand, there is a stream of research focusing on binary understanding models [60, 57], which target learning nuanced code semantics with structures of low-level code, which is critical for software security. Both fields are evolving through continuous
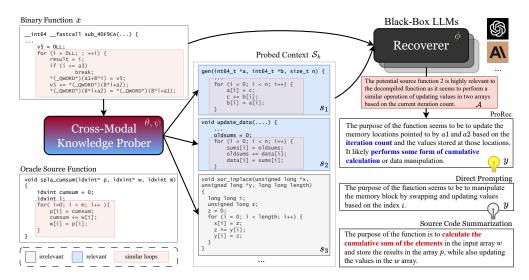
---

[*]Corresponding author.

Figure 1: The `ProRec` Framework for human-oriented binary reverse engineering. The figure shows a simple example of lifting a cumsum function from binary to human readable summarization. The probed contexts synthesized by the cross-modal knowledge prober, while not identical to the oracle source code of the query binary, exhibit informativeness in terms of symbol names and correct loop structure. These contexts help the black-box LLMs to successfully recover the high-level functionality of binary function in the summary that is consistent with the source code summary, moving beyond merely describing its low-level operations.

pre-training on expansive datasets of uni-modal data, setting new benchmarks in both effectiveness and complexity.

Human-Oriented Binary Reverse Engineering (HOBRE), which involves automatically lifting binary to human understandable contents [14, 68], typically source-code related, occupies a unique intersection between these two fields. Existing decompilers for binary reverse engineering are able to translate binary code to C-style code that is functionally equivalent to its original source code. However, a significant semantic gap remains between the decompiled code and its original source, primarily due to the absence of meaningful symbolic information in binary code. Hence, human expertise is still indispensable in the reverse engineering process. HOBRE aims to bridge this semantic gap, which traditionally requires substantial human effort, by leveraging cross-modal deep learning models. Existing approaches either train task-specific small expert models in a supervised manner [32, 1, 68], which lack generalizability as shown in later evaluations [56], or require extensive continual pre-training of uni-modal SCFMs [30] which is undesirable considering cost and the risk of forgetting previously acquired source code knowledge [34, 72]. There are also attempts in directly prompting LLMs for HOBRE, which, even though demonstrates better generalizability than small supervised models, also face challenges in understanding stripped decompiled code that lacks symbolic information [31].

Our insight is that this semantic gap between binary and source code is analogous to the gap between low-level pixels in images and high-level concepts in natural language, which can be bridged with sufficient understanding of both. Inspired by the achievements of multi-modal models that seamlessly integrate vision, audio, or other signals with language to facilitate reasoning [2, 39, 42, 45], we hypothesize that HOBRE could similarly benefit from leveraging uni-modal models developed for both source code and binary code. Such integration would enhance our ability to bridge the semantic gap and enable more effective semantic lifting.

In this paper, we validate this idea by proposing a novel probe-and-recover framework `ProRec` that incorporates a binary-source encoder-decoder model and black-box LLMs for HOBRE, featuring a compute-efficient cross-modal alignment approach of a binary function encoder and a frozen SCFM for the binary-source model. The workflow of `ProRec` is shown in Figure 1. The aligned binary-source model acts as a *cross-modal-knowledge-prober* that can synthesize symbol-rich, diverse source

code fragments condition on binary input, denoted as *probed contexts*. The black-box LLM functions as *recoverer* that takes as input the binary function together with the probed contexts for tasks such as binary summarization. Intuitively, the conditional source code synthesis by the aligned binary-source code model can be viewed as probing the base SCFM as a parametric knowledge base [52] with a binary function as query, given that the SCFM's weights remains unchanged before and after the alignment. A black-box LLM analyzes and aggregates these knowledgable contexts with the binary function for recovery. This way, `ProRec` leverages both cross-modal aligned knowledge and strong reasoning ability of LLMs and can outperform directly letting the LLM to reason. `ProRec` is general and can be applied to different base architectures, continually evolve with base models.

We demonstrate the effectiveness of `ProRec` on two core tasks in reverse engineering [9, 10]: binary summarization and binary function name recovery. The former aims to generate natural language descriptions for a binary function, and the later aims to recover the function name of a decompiled function. We evaluate `ProRec` on a diversified dataset compiled from GitHub repositories, demonstrating improvements of 3.1% (10.3% relative gain) in CHRF and 12% (16.7% relative gain) in a GPT4-based metric that has high correlation with human judgement on the summarization task over zero-shot baseline. We conduct human study to show the effectiveness of the newly proposed GPT4-based metric. On name recovery tasks, `ProRec` significantly improves over zero-shot baseline by 6.7% and 7.4% for token-level precision and recall, respectively. For both tasks, `ProRec` also consistently show advantage over a retrieval-augmented baseline with a strong cross-modal dense retriever. [2]

## 2  `ProRec`: Reverse Binary by Probing Source Code Foundation Models

In this section, we first present the `ProRec` framework in §2.1. Next, we describe the neural architecture used for the cross-modal knowledge prober and recoverer in §2.2. The training for the prober in is detailed in §2.3, followed by the comphrehensitve explanation of the knowledge probing stage in §2.4.

**Formulation**  Given a binary file, we can leverage binary analysis tools [3] to obtain each binary function $x$. Specifically, $x$ can either be in its disassembled code form which we denote as $x_{\texttt{asm}}$, or its stripped decompiled code form, denoted as $x_{\texttt{dec}}$. $x_{\texttt{asm}}$ and $x_{\texttt{dec}}$ are semantically equivalent and similarly unreadable. The goal is to recover human readable information $y$ given $x_{\texttt{dec}}$ and $x_{\texttt{asm}}$.

### 2.1  The Probe-and-Recover Framework

`ProRec` assumes a binary understanding model parameterized by $\theta$, an open-source SCFM parameterized by $\psi$, and a black-box LLMs by $\phi$. As illustrated in Figure 1, the binary model together with the SCFM form the cross-modal knowledge prober. The black-box LLM serves as a recoverer. The cross-modal prober can synthesize source code fragments given binary input. The recoverer takes in augmented context with binary code to analyze and perform final recovery.

Conceptually, the `ProRec` framework decomposes the probability to generate $y$ into three parts, the probability of a set of $k$ source code fragments $\mathcal{S}_k = \{s_1, \cdots, s_k\}$ being relevant to input $P(\mathcal{S}_k|x)$, the probability of LLM's relevance analysis of the source code fragments $P(\mathcal{A}|\mathcal{S}_k, x)$, and the probability of generating the recovery results conditioned on the analysis and source code fragments.

$$P(y|x) = \sum_{\mathcal{S}_k \sim P_{\theta,\psi}(\cdot|x), \mathcal{A} \sim P_\phi(\cdot|\mathcal{S}_k,x)} P_\phi\left(y \big| \mathcal{A}, \mathcal{S}_k, x\right) \cdot P_\phi\left(\mathcal{A}|\mathcal{S}_k, x\right) \cdot P\left(\mathcal{S}_k \big| x\right) \quad (1)$$

The decomposition is similar to that of retrieval-augmented generation [38, 67], where $p(y|x) = \sum_{s \in \text{top-}k(S^*)} P(y|s, x)P(s|x)$, given a document pool $S^*$. However, there are two major differences. First, the source code fragments $\mathcal{S}_k$ are not retrieved from $S^*$, instead, they are sampled from the conditional distribution of the prober $P_{\theta,\psi}(\cdot|x)$. Due to the alignment strategy (discussed in §2.3), source code fragments sampled from the prober's distribution have more flexibility than those
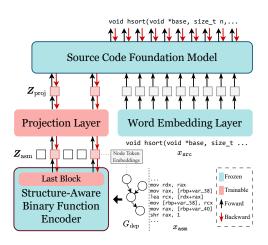
---

[3]`https://hex-rays.com/ida-pro/`

Figure 2: The prober architecture and compute-efficient alignment with limited trainable parameters.
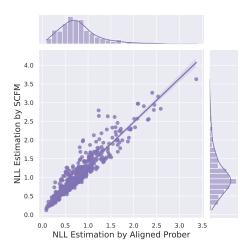


Figure 3: Average negative log-likelihood of source function tokens estimated by base SCLM and that of source function conditioned on its binary counterpart estimated by the aligned prober.

retrieved a fixed document pool in binary reverse engineering scenario, potentially less noisy. We empirically demonstrate the superiority of probing over retrieval for augmentation in §4.

Second, we stress the internal analysis from the LLM denoted as $P_\phi(\mathcal{A}|\mathcal{S}_k, x)$ in the decomposition. The insight is that, even though high-level recovery requires additional domain information to hint black-box LLMs for further induction, that doesn't necessarily mean LLMs totally lacks of such knowledge (since proprietary LLMs can be significantly larger than open-source code language models in size and may have more training data, just different mixtures), it might be some long-tail knowledge that requires better prompting to exploit [37, 71]. On the other hand, the analysis help LLMs to be less influenced by the noisy contexts. This is beneficial for both retrieved and probed contexts.

Note that, in Equation 1, the final probability is marginalized over all possible $\mathcal{S}_k$ and $\mathcal{A}$. In practice, we take the most probable $\mathcal{S}_k$ and keep the analysis with in the response before final result, without heavy sampling. We will discuss the sampling of each $s$ within $\mathcal{S}_k$ in §2.4.

## 2.2 Model Architecture and Instantiation

`ProRec` is a general framework and is not bounded to existing models and architectures. This section is our current implementation of the prober and recoverer that provide the best performance in our experiments.

**Cross-Modal Knowledge Prober** The core of `ProRec` is the cross-modal prober, which is an encoder-decoder model aligned in the token embedding space of the SCFM, as illustrated in Figure 2. We would like to benefit from both pre-trained binary function encoders that possesses binary domain knowledge and the strong SCFMs for generalizable probing. We choose the state-of-the-art CODEART [57] as our structure-aware binary function encoder $g(\cdot)$. CODEART is a BERT-like transformer encoder that takes as input a disassembled binary function $x_{\texttt{asm}}$ along with its dependency graph $G_{\text{dep}}$ obtained by program analysis, and outputs the embeddings for all assembly code tokens and graph node tokens (each graph node token corresponds to one instruction, e.g., `mov rax, [rbp+var_40]`, in the assembly code). We choose the Code-Llama [55] family as our base SCFM [4].

---

[4]We also tried other SCFMs like DeepSeek-Coder [22] or StarCoder2 [43] in our preliminary study and find that Code-Llama performs best as a base SCFM for our prober.

For the final prober architecture, we apply a simple two-layer MLP to project the *node token embeddings* $\boldsymbol{Z}_{\text{asm}}$, with indices N_IDX in all token embeddings, to source code token embeddings space.

$$\boldsymbol{Z}_{\text{asm}} = \text{CODEART}\left(x_{\text{asm}}, G_{\text{dep}}\right)[\text{N\_IDX}, :] \in \mathbb{R}^{l_n \times d_b}, \; \boldsymbol{Z}_{\text{proj}} = \text{MLP}(\boldsymbol{Z}_{\text{asm}}) \in \mathbb{R}^{l_n \times d_s} \quad (2)$$

where $l_n$ denotes the number of node tokens, $d_b$ is the dimension of the binary encoder and $d_s$ is the dimension of the SCFM. The projected embeddings are fed into the SCLM as an additional prefix before regular subtoken embeddings for conditional generation.

We only use node token embeddings as binary features due to their significantly smaller quantity compared to all token embeddings (approximately one eighth) since assembly functions tend to be long. These embeddings also already capture some structural abstraction of binary code which is meaningful in HOBRE tasks.

**Recoverer** We leverage proprietary black-box LLMs (GPT3.5, Claude-3, and Gemini-1.0) as our recoverer, since they have strong reasoning ability and support long contexts. Specifically, the LLMs are prompted with $x_{\text{dec}}$ as zero-shot baseline. For retrieval-augmented baseline and ProRec, we append the additional context to the original input and instruct LLMs to analyze relevance and then generate recovery. Detailed prompts can be found in Appendix C.

### 2.3 Prober Training

The training of prober contains two stages: the pre-alignment of the binary encoder to a source code encoder, and the binary encoder-SCFM alignment. Both utilize data in the form of paired binary-source functions. The goal is to gradually align the binary encoder with the base SCFM with minimum knowledge loss.

**Contrastive Assembly-Source Code Pre-Alignment** Since CODEART is exclusively pre-trained on binary code corpus [57], we first align it with a pre-trained source code encoder codet5p-embedding-110m [63] in the function-level embedding space as a pre-alignment stage in order to facilitate the later encoder-decoder alignment. To achieve this, we add a projection head for each encoder to project their [CLS] token embeddings to the same dimension $d_{\text{enc}}$, forming a standard dual-encoder [29]. This dual-encoder can encode $(x_{\text{asm}}, G_{\text{dep}})$ into $\boldsymbol{h}_{\text{asm}} \in \mathbb{R}^{d_{\text{enc}}}$ and $x_{\text{src}}$ into $\boldsymbol{h}_{\text{src}} \in \mathbb{R}^{d_{\text{enc}}}$. We train the dual-encoder in a CLIP-like symmetric contrastive fashion [54]. Since the implementation is relatively standard, we refer readers to Appendix A for details.

The dual-encoder can function as a dense retriever to rank and retrieve the top-$k$ source functions from the source function pool of the training set for a query binary function, based on the similarity measure $\text{sim}(x_{\text{asm}}, x_{\text{src}}) = \cos(\boldsymbol{h}_{\text{asm}}, \boldsymbol{h}_{\text{src}})$. It achieves 84% recall@1 on validation set with a pool of 10k examples, demonstrating strong performance as a retriever. We utilize this dual-encoder to set up a retrieval-augmented recovery baseline to compare with ProRec in §4.

**Compute-Efficient Cross-Modal Prober Alignment** For encoder-decoder alignment, we freeze all the parameters within the SCFM because we intend to explore the extreme end of probing knowledge from it. We freeze CODEART from the first stage except for the last layer which is a transformer block for fast convergence and avoid too much change in the representation. The MLP is fully trainable. The objective of the alignment is to maximize

$$P(x_{\text{src}}|x_{\text{asm}}, G_{\text{dep}}) = \prod_i^{|x_{\text{src}}|} P_{\theta, \psi}(x_i|\boldsymbol{Z}_{\text{proj}}, x_{<i}) \quad (3)$$

The limited amount of trainable parameters results in efficient training. For memory efficiency, we apply quantization (4bit or 8bit) [17, 18] to the base SCFM during alignment.

One evidence that the knowledge of the aligned prober is mainly from the SCFM pre-training instead of learned during alignment is shown in Figure 3. We sampled 500 $(x_{\text{asm}}, x_{\text{src}})$ pairs from the validation set and find that the average negative log-likelihood $-\frac{1}{|x_{\text{src}}|} \log P_\psi(x_{\text{src}})$ for $x_{\text{src}}$ provided

by the base SCFM and $-\frac{1}{|x_{\text{src}}|} \log P_{\theta,\psi}(x_{\text{src}}|x_{\text{asm}}, G_{\text{dep}})$ for $x_{\text{src}}$ conditioned on the $x_{\text{asm}}$ provided by the aligned prober are highly correlated, indicating that the prober's ability is consistent with the base SCFM. Another interesting observation is that, instruction-tuned SCFMs typically show higher losses during alignment than their original models, which also implies the significance of pre-trained knowledge of source code for cross-modal ability as instruction-tuning may cause forgetting.

## 2.4 Cross-Modal Knowledge Probing

For the probing process, i.e., sampling $\mathcal{S}_k$ with the aligned $P_{\theta,\psi}(\cdot|x_{\text{asm}}, G_{\text{dep}})$, we want to cover a diverse yet relevant set of candidates. We leverage nucleus sampling [26] to first let the prober generate a relatively large set of source function signatures with high randomness (top-$p = 0.75$). We use idea similar to retrieval by training a binary-signature dual-encoder to rank generated signatures and filter out the noisy ones. Ultimately, we use the prober to further complete the remaining signatures with smaller randomness (top-$p = 0.5$). Since signature is short and important for HOBRE, our strategy achieves both better relevance compared to using a fixed small $p$ for full function generation and better efficiency compared to sampling a large set of functions with a large $p$.

# 3 Experiment Setup

We evaluate `ProRec` on two binary reverse engineering tasks: recovering function names from decompiled code (§3.2), and summarizing function semantics from decompiled code (§3.1). In this section, we first introduce our dataset, and the setups of each task.

**Dataset**  The training and evaluation of `ProRec` requires pair-wise data between a binary function and its corresponding source code. To the best of our knowledge, there is no publicly available dataset that contains matched source code with the binary program. Therefore, we follow a widely adapted practice in the reverse engineering domain [13, 36, 7], using GHCC [5] to automatically clone and compile repositories from GitHub. After the compilation, we map the resulting binary programs with the corresponding source code functions leveraging the debug information in binary programs. In total, our data consists of 270k pairs of binary and source code functions. We split 260k data samples for training and 10k data samples for test. We use 5% of the training data as the validation dataset. To make the evaluation cost tractable, we randomly sample 1k samples from the test dataset. For details in data processing and quality assurance, please see Appendix B.1.
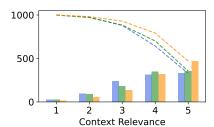
## 3.1 Binary Summarization

A binary summarization tool takes as input a snippet of decompiled code, and outputs natural language descriptions. It facilitates two key processes in the reverse engineering practice [9]: understanding the purpose of a program, and understanding the functionality of a program. Please see Appendix E for a detailed example.

**Setup**  We use the summary for a *source code* function as the reference summary for the corresponding decompiled code, following existing work [56]. We instruct an LLM to summarize decompiled code with three setups: (1) providing the model with only the decompiled code; (2) additionally provide the relevant source code snippets retrieved by a retriever; (3) additionally provide and the source code snippets generated by `ProRec`. The first two setups are considered baseline approaches for comparison.

**Metrics**  We use CHRF [53] instead of other commonly used metrics in summarization such as BLEU [48]. Our user study in Appendix D shows that only CHRF is consistent with human preferences. We additionally propose GPT4-based metrics for binary summarization, since recent studies show that LLM-based metrics show higher consistency with human judgement [73]. We ask GPT4 to measure the context and functionality relevance of a summary. The questions are derived from a thorough study on human reverse engineers [9]. Details are in Appendix B.2. The proposed metrics show high correlation with human judgement. Details are in Appendix D.
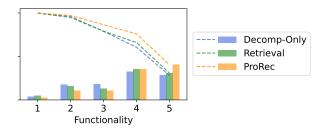
---

[5] https://github.com/huzecong/ghcc

Figure 4: Scores from GPT4 evaluator for summaries of GPT3.5-turbo. The x-axes denote context relevance (left) and functionality (right), respectively. Larger scores are better. Bars denote the number of summaries with the corresponding score, and dashed lines denote the number of summaries with *at least* the corresponding score.

## 3.2  Binary Function Name Recovery

We further evaluate the performance of `ProRec` on the binary function name recovery task. Different from generating summary for a decompiled function, recovering function name requires a tool to have more accurate understanding about program contexts and more concise abstraction for program semantics, as shown by our case study in Appendix E. Please refer to Appendix B.3 for the importance and use scenarios of function name recovery.

**Setup**    We use the source code function names as the ground truth for name recovery. Similar to the binary summarization task, we conduct experiments with three setups: we input to an LLM the decompiled code, the decompiled code with source code snippets obtained by a retriever, and the decompiled code with source code snippets generated by `ProRec`. The first two setups are considered baselines for comparison.

**Metrics**    We use two sets of metrics to evaluate function name. We adapt ROUGE-L [8]for function name since it provides a fine-grained evaluation on the n-gram level. Following existing work in the reverse engineering domain [32], we further evaluate function names by tokenizing pairs of predicted names and ground truth names, and calculating set intersections between the token sets of the predictions and the ground truth. Please see Appendix B.4 for formal definition. For both metrics, we report precision and recall.

## 4  Results

In all the following experiments, we report `ProRec` results based on CodeLlama-34b (4bit quantized). For both the retrieval-augmented baseline (+retrieval) and `ProRec` (+ProRec), we use their top-5 contexts as augmentation. The versions of the black-box LLMs are `gpt-3.5-turbo-1106` for GPT3.5-turbo, `claude-3-haiku-20240307` for Claude-3, `gemini-1.0-pro` for Gemini-Pro, and `gpt-4-turbo-2024-04-09` for GPT4 Evaluator.

### 4.1  Binary Summarization Results

We show the results for binary summarization in Table 1. Observe that `ProRec` helps all models generate better summary in terms of CHRF. A retriever, on the other hand, may introduce noise to a model and even makes the results worse (e.g., for the Gemini-Pro model). Moreover, we can see that `ProRec` achieves higher scores when evaluated with the GPT4Evaluator on functionality (G4-F) and context relevance (G4-C), indicating the summary of `ProRec` is more helpful to a human reverse engineer.

We further analyze the results of GPT4Evaluator to illustrate the advantage of `ProRec`. The results for summaries generated by GPT-3.5 are visualized in Figure 4. It is worth-noting that we define the score 3 as a "neutral" score, meaning that a summary does not contain specific context (for the context relevance question) or contains only correct but low-level operations without high-level abstractions (for functionality question). We can see that for most cases, GPT-3.5 achieves a score with at least 3.

Table 2: Binary function name recovery results. The following three columns denote the precision, recall, and F1 score proposed by the reverse engineering work SymLM [32]. The last three columns denote the precision, recall, and F1 score for ROUGE-L [8].

| | | SymLM | | | ROUGE-L | | |
|---|---|---|---|---|---|---|---|
| | | Precision | Recall | F1 | Precision | Recall | F1 |
| GPT-3.5-turbo | - | 16.3 | 20.9 | 17.2 | 39.8 | 54.7 | 44.8 |
| GPT-3.5-turbo | +retrieval | 15.5 | 21.3 | 17.0 | 39.4 | 56.5 | 45.2 |
| GPT-3.5-turbo | +ProRec | **22.2** | **28.3** | **23.5** | **41.7** | **58.3** | **47.3** |
| Gemini-Pro | - | 25.3 | 28.7 | 25.3 | 44.5 | **55.3** | 47.8 |
| Gemini-Pro | retrieval | 22.7 | 23.3 | 21.6 | 44.2 | 50.2 | 45.5 |
| Gemini-Pro | +ProRec | **32.6** | **30.5** | **29.9** | **50.8** | 53.8 | **50.6** |
| Claude-3 | - | 16.5 | 22.3 | 17.9 | 38.9 | 48.9 | 40.3 |
| Claude-3 | retrieval | 19.9 | 23.9 | 20.5 | 40.6 | 51.1 | 43.3 |
| Claude-3 | +ProRec | **25.9** | **29.4** | **26.0** | **44.6** | **54.7** | **47.4** |

That indicates the LLM can largely understand the low-level behaviors of decompiled code. That is because decompiled code is in the C-syntax.

On the other hand, we can see that for the context relevance question, both RAG and `ProRec` introduces more useful context information to the model, and thus the resulting summaries have closer relevance to the groundtruth source code. Especially, queries with the code snippets generated by `ProRec` achieve more scores 4 and 5 than queries enhanced with a retriever's results. That illustrates `ProRec` indeed generates code snippets with better context relevance than a retriever.

For the functionality question, we can observe similar patterns. That indicates better contexts introduced by `ProRec` help the LLM to understand code functionality. We show a detailed example in Appendix E.

## 4.2 Binary Function Name Recovery Results

We show results for binary function name recovery in Table 2. We can see that the code snippets generated by `ProRec` helps all three LLMs predict better names in terms of precision, recall, and F-1 score measured by the metrics designed for reverse engineering task [32]. Especially, `ProRec` outperforms a retriever by a large margin, indicating that `ProRec` generates more relevant code than a retriever. For Rouge-L-based metrics, `ProRec` introduces better performance for GPT-3.5 and Claude-3. However, for Gemini-Pro, the recall is slightly lower than directly query the model without relevant source code snippets. We find that is because Gemini-Pro tends to generate

Table 1: Binary summarization results. G4-F and G4-C denote GPT4Evaluator for functionality and context relevance, respectively.

| Model | | CHRF | G4-F | G4-C |
|---|---|---|---|---|
| GPT-3.5-turbo | - | 30.4 | 3.6 | 3.8 |
| GPT-3.5-turbo | +retrieval | 31.7 | 3.7 | 3.9 |
| GPT-3.5-turbo | +ProRec | **33.5** | **4.2** | **4.0** |
| Gemini-Pro | - | 27.1 | 3.7 | 3.5 |
| Gemini-Pro | +retrieval | 26.4 | 3.4 | 3.2 |
| Gemini-Pro | +ProRec | **27.6** | **3.8** | **3.6** |
| Claude-3 | - | 33.5 | 3.7 | 3.9 |
| Claude-3 | +retrieval | 33.9 | 3.8 | 3.9 |
| Claude-3 | +ProRec | **34.9** | **4.0** | **4.1** |

lengthy descriptive names when not provided with specific contexts. And vice versa, the predicted name becomes shorter when provided with specific contexts. For example, without specific context, Gemini-Pro predicts `printf_with_variable_arguments` for a function printing error messages. After getting `ProRec`'s results, the prediction becomes `error`. That means, without context information, Gemini-Pro has higher probability to coincide with the groundtruth tokens since it generates longer names. On the other hand, the precision is significantly lower than ones with better context.
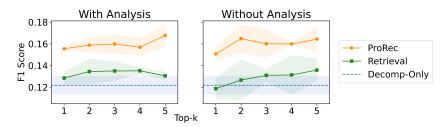
8

Figure 5: Binary function name recovery results with and without LLM's internal analysis by using top-$k$ additional contexts on 100 examples.

Table 3: Statistics of Prober with Different Base SCFM Sizes

| Base SCFM | Trainable Params | Ratio (%) | Eval Loss | N-gram Recall (1-4) | CHRF |
|---|---|---|---|---|---|
| CodeLlama-7b | 27M | 0.393 | 0.6756 | 27.22 / 13.69 / 8.21 / 5.14 | 31.52 ($\pm$0.642) |
| CodeLlama-13b | 37M | 0.283 | 0.6387 | 27.51 / 13.88 / 8.40 / 5.32 | 32.01 ($\pm$0.886) |
| CodeLlama-34b | 80M | 0.237 | 0.5786 | 27.58 / 14.06 / 8.55 / 5.45 | 31.54 ($\pm$0.163) |

## 5 Analysis

### 5.1 How does black-box LLM's internal analysis $\mathcal{A}$ help robust recovery?

We study the influence of LLM's internal analysis by evaluating retrieval-augmented recovery and `ProRec` with different number of additional contexts, since we believe this kind of internal analysis is crucial for LLM-based recoverers to perform robust binary recovery with additional contexts, especially when the provided contexts are noisy. We run binary function name recovery for 100 randomly sampled examples, 3 times, for zero-shot recovery (`decomp-only`), retrieval-augmented recovery (`retrieval`), and `ProRec`. As shown in Figure 5, the internal analysis consistently reduce the variance of function name recovery performance of both retrieval-augmented recovery and `ProRec`. This is particularly true for retrieval when $k$ gets large. We deem that it may due to a lack of function-level similar source code in the data store. On the other hand, we observe sometimes LLM tend to be too conservative without leveraging the extra contexts with the internal analysis, potentially because of our not specifically optimized prompts which can be fixed by making some adjustments. Moreover, we argue that *misleading is worse than not informative*, and reverse engineers can further interact with LLMs for more aggressive induction after obtaining a conservative response.

### 5.2 Ablation Study on Base Source Code Foundation Model Size

`ProRec`'s performance relies on the ability of the base SCFM, where size is a crucial indicator since knowledge is represented by model parameters. Therefore, we study the influence of base SCFM size. We train three probers based on CodeLlama-7b, CodeLlama-13b, and CodeLlama-34b, all in 4bit quantization for fair comparison. We report statistics of these three probers in Table 3. As shown in the table, with growing number of base model size, the prober achieve a lower loss on validation set, which leads to an increase in average n-gram overlap of probed source code fragments and the oracle source function, which we run 3 times on 100 examples for each row. However, n-gram overlap with oracle source function seems not to significantly influence downstream task performance like CHRF for binary summarization. We hypothesize that this is potentially due to the tasks like binary summarization is not very sensitive to subtle symbolic difference, which means we can leverage modest size SCFM for probing instead of large ones, being economic in real practice.

### 5.3 Correlation Between Human Preference and Auto-Metrics on Binary Summarization

For the binary summarization task, we conduct a human study to measure the correlation between human preference and automatic metrics. We conclude that the automatic metrics CHRF and the GPT4Evaluator are more consistent with human preferences. Please refer to Appendix D for details.

9

# 6 Related Work

## 6.1 Large Multimodal Models

Recent advancements in vision-language models have demonstrated their efficacy across a range of practical applications such as image captioning [69], visual question answering [5, 16, 3], and image-text matching [40]. While the limited availability of datasets that align different modalities was perceived as a major impediment to scalability, recent works leverage the knowledge embedded within pre-trained large language models, including Flamingo [2], OpenFlamingo [6], Palm-E [19], BLIP-2 [39], along with other existing works [15, 41, 75, 21]. Beyond their capacity to interpret diverse information modalities such as images [62] and audio [27], LLMs have increasingly been aligned with graph structures [11, 58] and gained widespread attention. In particular, there have been successful attempts that leverage LLMs for graph data involves the Graph2Text strategy, which transforms graph data into textual representations. This technique has been effectively utilized in several studies [61, 23, 70]. The designs in the prober of `ProRec` share some similarity with recent LMMs, with a modality-specific encoder, and a SCFM decoder. However, we tackle the binary-source code multi-modality which is largely unexplored compared to popular modalities. Also, the multi-modal prober is used in a larger probe-and-recover framework instead of end-to-end training.

## 6.2 Retrieval-Augmented Generation

Retrieval-augmented generation is widely applied in knowledge-intensive scenarios, such as question answering [24, 28, 33], molecule generation [65], and source code generation [74]. By leveraging a non-parametric datastore, retrieval-augmented approaches decompose knowledge and LMs, can complement some long-tail knowledge or keep the knowledge up-to-date without heavy tuning the model which is costly. `ProRec`, on the other head, try to exploit knowledge within a parametric SCLM for black-box LLM-based binary recovery. A closely related work is GENREAD [71] that prompts InstructGPT to generate context instead of retrieval for knowledge intensive tasks. `ProRec` differs from this work in that binary recovery requires cross-modal understanding and informative contexts cannot be obtained by directly prompting LLMs We introduce specially designed cross-modal alignment to allow informative context generation.

## 6.3 Binary Reverse Engineering

Advances in machine learning models have been widely used to solve challenging tasks in binary program analysis [51, 49, 60, 57, 50]. However, most work focuses on reasoning binary program, and is not human-oriented. Another stream of work trained smaller end-to-end models for individual human oriented tasks, such as variable name prediction [47, 13] and function name recovery [32]. Nonetheless, these models are not benefiting from pretraining efforts and thus have sub-optimal performance [56]. Preliminary study shows HOBRE remains a challenge for state-of-the-art LLMs [31, 56]. Our efforts attempt to address this challenge, leveraging pre-trained knowledge of binary understanding models and SCFMs to help HOBRE.

# 7 Conclusion

In this paper, we introduced a novel probe-and-recover framework, `ProRec`, designed to bridge the semantic gap between binary code and human-understandable source code. By integrating an aligned binary-source encoder-decoder model with black-box large language models, our approach effectively synthesizes symbol-rich code fragments from binary input, providing valuable context for improving binary analysis tasks. Our extensive evaluations demonstrate that `ProRec` significantly enhances performance in both binary summarization and binary function name recovery tasks.

**Limitations & Future Work**   We experiment with a simple achitecture and straightforward alignment of the binary-source prober in this paper, which might not be optimal for `ProRec`. Future work can explore better prober architecture and alignment objectives. Moreover, currently we only focus on intra-procedure analysis, similar to most existing work. In practice, HOBRE needs to deal with full binary with multiple functions. An important direction will be extending `ProRec` to inter-procedure scenarios, where additional information from the whole program such as call-graph can be leveraged.

# References

[1] 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.

[2] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems*, 35:23716–23736, 2022.

[3] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton Van Den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3674–3683, 2018.

[4] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[5] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

[6] Anas Awadalla, Irena Gao, Josh Gardner, Jack Hessel, Yusuf Hanafy, Wanrong Zhu, Kalyani Marathe, Yonatan Bitton, Samir Gadre, Shiori Sagawa, et al. Openflamingo: An open-source framework for training large autoregressive vision-language models. *arXiv preprint arXiv:2308.01390*, 2023.

[7] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

[8] Marcello Barbella and Genoveffa Tortora. Rouge metric evaluation for text summarization techniques. *Available at SSRN 4120317*, 2022.

[9] Adam R Bryant. *Understanding how reverse engineers make sense of programs from assembly language representations*. Air Force Institute of Technology, 2012.

[10] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2765–2782, 2022.

[11] Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. Graphllm: Boosting graph reasoning ability of large language model. *arXiv preprint arXiv:2310.05845*, 2023.

[12] 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.

[13] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, Boston, MA, August 2022. USENIX Association.

[14] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[15] Wenliang Dai, Junnan Li, Dongxu Li, Anthony Meng Huat Tiong, Junqi Zhao, Weisheng Wang, Boyang Li, Pascale N Fung, and Steven Hoi. Instructblip: Towards general-purpose vision-language models with instruction tuning. *Advances in Neural Information Processing Systems*, 36, 2024.

[16] Abhishek Das, Satwik Kottur, Khushi Gupta, Avi Singh, Deshraj Yadav, José MF Moura, Devi Parikh, and Dhruv Batra. Visual dialog. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 326–335, 2017.

[17] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[18] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[19] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

[20] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[21] Peng Gao, Jiaming Han, Renrui Zhang, Ziyi Lin, Shijie Geng, Aojun Zhou, Wei Zhang, Pan Lu, Conghui He, Xiangyu Yue, et al. Llama-adapter v2: Parameter-efficient visual instruction model. *arXiv preprint arXiv:2304.15010*, 2023.

[22] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[23] Jiayan Guo, Lun Du, and Hengyu Liu. Gpt4graph: Can large language models understand graph structured data? an empirical evaluation and benchmarking. *arXiv preprint arXiv:2305.15066*, 2023.

[24] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.

[25] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[26] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2019.

[27] Rongjie Huang, Mingze Li, Dongchao Yang, Jiatong Shi, Xuankai Chang, Zhenhui Ye, Yuning Wu, Zhiqing Hong, Jiawei Huang, Jinglin Liu, et al. Audiogpt: Understanding and generating speech, music, sound, and talking head. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 23802–23804, 2024.

[28] Gautier Izacard and Édouard Grave. Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 874–880, 2021.

[29] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. Binaryai: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[30] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Nova $^{+}$: Generative language models for binaries. *arXiv preprint arXiv:2311.13721*, 2023.

[31] 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.

[32] 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.

[33] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, 2020.

[34] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[35] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

[36] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

[37] Yoav Levine, Itay Dalmedigos, Ori Ram, Yoel Zeldes, Daniel Jannai, Dor Muhlgay, Yoni Osin, Opher Lieber, Barak Lenz, Shai Shalev-Shwartz, et al. Standing on the shoulders of giant frozen language models. *arXiv preprint arXiv:2204.10019*, 2022.

[38] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[39] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International conference on machine learning*, pages 19730–19742. PMLR, 2023.

[40] Kunpeng Li, Yulun Zhang, Kai Li, Yuanyuan Li, and Yun Fu. Visual semantic reasoning for image-text matching. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 4654–4662, 2019.

[41] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning. *arXiv preprint arXiv:2310.03744*, 2023.

[42] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36, 2024.

[43] 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.

[44] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

[45] Seungwhan Moon, Andrea Madotto, Zhaojiang Lin, Tushar Nagarajan, Matt Smith, Shashank Jain, Chun-Fu Yeh, Prakash Murugesan, Peyman Heidari, Yue Liu, et al. Anymal: An efficient and scalable any-modality augmented language model. *arXiv preprint arXiv:2309.16058*, 2023.

[46] OpenAI. Gpt-4 technical report, 2024.

[47] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. "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 152–152. IEEE Computer Society, 2024.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[49] 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.

[50] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. In *NDSS*. The Internet Society, 2021.

[51] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[52] Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. Language models as knowledge bases? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2463–2473, 2019.

[53] Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the tenth workshop on statistical machine translation*, pages 392–395, 2015.

[54] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[55] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[56] 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 preprint arXiv:2404.09836*, 2024.

[57] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. Codeart: Better code models by attention regularization when symbols are lacking. *arXiv preprint arXiv:2402.11842*, 2024.

[58] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. Graphgpt: Graph instruction tuning for large language models. *arXiv preprint arXiv:2310.13023*, 2023.

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

[60] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jtrans: Jump-aware transformer for binary code similarity. *arXiv preprint arXiv:2205.12713*, 2022.

[61] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36, 2024.

[62] Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, et al. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. *Advances in Neural Information Processing Systems*, 36, 2024.

[63] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[64] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[65] Zichao Wang, Weili Nie, Zhuoran Qiao, Chaowei Xiao, Richard Baraniuk, and Anima Anand-kumar. Retrieval-based controllable molecule generation. In *The Eleventh International Conference on Learning Representations*, 2022.

[66] Jiayi Wei, Greg Durrett, and Isil Dillig. Coeditor: Leveraging contextual changes for multi-round code auto-editing. *arXiv preprint arXiv:2305.18584*, 2023.

[67] Shi Weijia, Min Sewon, Yasunaga Michihiro, Seo Minjoon, James Rich, Lewis Mike, et al. Replug: Retrieval-augmented black-box language models. *ArXiv: 2301.12652*, 2023.

[68] 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.

[69] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.

[70] Ruosong Ye, Caiqi Zhang, Runhui Wang, Shuyuan Xu, and Yongfeng Zhang. Natural language is all a graph needs. *arXiv preprint arXiv:2308.07134*, 2023.

[71] Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, Soumya Sanyal, Chen-guang Zhu, Michael Zeng, and Meng Jiang. Generate rather than retrieve: Large language models are strong context generators. In *The Eleventh International Conference on Learning Representations*, 2022.

[72] Yuexiang Zhai, Shengbang Tong, Xiao Li, Mu Cai, Qing Qu, Yong Jae Lee, and Yi Ma. Investigating the catastrophic forgetting in multimodal large language models. *arXiv preprint arXiv:2309.10313*, 2023.

[73] 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, 2024.

[74] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2022.

[75] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. Minigpt-4: En-hancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.

# A    Details for Training Assembly-Source Code Dual-Encoder

We discuss details of the contrastive training of the assembly-source code dual-encoder in this section. Given a mini-batch $\mathcal{B} = \{(x_i^a, x_i^s)\}_{i=1}^N$ with batch size $N$, where $x_i^a$ represents the $i$-th tuple of assembly code and its dependency graph, $x_i^s$ represents the corresponding $i$-th source code, we train the dual-encoder, $g(\cdot)$ for binary encoder, and $h(\cdot)$ for source encoder, with the following objective

$$\mathcal{L}_{\text{dual-enc}} = \frac{1}{2}(\mathcal{L}_{\text{a2s}} + \mathcal{L}_{\text{s2a}}) \tag{4}$$

where

$$\mathcal{L}_{a2s} = \sum_{i=1}^{N} -\log \frac{\exp(\text{sim}(g(x_i^a), h(x_i^s)))}{\sum_{j=1}^{N} \exp(\text{sim}(g(x_i^a), h(x_j^s)))} \tag{5}$$

and

$$\mathcal{L}_{s2a} = \sum_{i=1}^{N} -\log \frac{\exp(\text{sim}(g(x_i^a), h(x_i^s)))}{\sum_{j=1}^{N} \exp(\text{sim}(g(x_j^a), h(x_i^s)))} \tag{6}$$

Here, we use cosine similarity for $\text{sim}(\cdot, \cdot)$. In order to have more negative samples, we use momentum encoders [25] for both modality with a queue size 4096. We train the model with learning rate 5e-5, a batch size of 16, 1k warmup steps, and 17k total steps.

The aligned dual-encoder as a cross-modal dense retriever achieves 84% recall@1 on the validation set with pool size 10k, which demonstrates that it has reasonable retrieval performance.

# B    Details in Experiment Setup

## B.1    Dataset Quality Assurance and Preprocessing

We ensure data quality by (1)selecting projects with no less than 20 stars, (2)including only executable binary programs that can be fully stripped, and (3)deduplicating our dataset by checking the source code string.

Initially, we obtained 18k projects from Github. We tried to compile all of them in x86-64 with O0, and discarded not compilable ones. It generates 106k executable binaries. We then match binary functions with source code functions by their function names, and deduplicate data samples by their source code strings (e.g., some utility functions may be used in multiple programs). Our final dataset containing 270k pairs of binary and source code functions.

## B.2    Details and Rationale for GPT4Evaluator

We provide GPT4 with the decompiled code, the corresponding source code, the reference summary, and the summary to evaluate. For each question, we adapt the Likert scale [6] and instruct GPT4 to output a score from 1(worst) to 5(best). We give detailed description for each score.

We derive our evaluator prompts from a thorough survey on reverse engineer [9]. The survey summarizes 8 sub-goals of human reverse engineers. We list them in Table 4 and categorize the goals into four scopes. We highlight ones that binary summarization can help.

Specifically, for goal (7), a reverse engineer aims to reason about the high-level abstraction of the program, e.g., what the program does, and how the program works [9]. We use the prompt in Figure 6 to evaluate how helpful a summary is to obtain the high-level picture of a program.

For goal (8), a reverse engineer reasons specific behavior individual functions to form the mental models [9] of the program logic. We use the prompt in Figure 7 to illustrate how accurate a summary is to describe the functionality of a program.

---

[6]https://en.wikipedia.org/wiki/Likert_scale

**A. Does the summary reflect relevant context (domain)? Answer the question in range 5(best) to 1(worst). Domain/context describes the purpose of a function. It is more of the general high-level domain (e.g., network, memory, CPS, physics, GUI, etc) rather than specific functionalities (e.g., sort, string comparison, memory allocation).**

- For 5, the summary and the reference should describe the same domain/context.

- For 4, the domain of the summary and the reference should be similar and relevant, although may not be exactly the same. The summary domain may be a superset or subset of the reference. The summary domain may be closely related to the reference domain. The summary and reference may be two different perspectives of a same specific domain.

- For 3, the summary does not explicitly mention a specific context. It only contains low level operations. From the summary, one cannot deduce the high-level purpose of the decompiled function.

- For 2, the summary is slightly misleading. The summary domain is different and not relevant to the reference domain. However, it is implied by the choice of words in the summary, and is not explicitly mentioned.

- For 1, the summary is completely misleading. The summary domain is irrelevant to the reference domain, and it is explicitly mentioned in the summary.

Your output should first briefly comment the summary from the aforementioned perspectives. Do not allow the length of the responses to influence your evaluation. Be as objective as possible.

Figure 6: Prompts for GPT4-Evaluator for asking context relevance.

**B. Does the summary reflect relevant functionality? Answer the question in range 1(best) to 5(worst). Functionality means the specific high-level behaviors performed in a function (e.g., sort, string comparison, decoding package, printing error messages).**

- For 5, the functionality in the summary should be almost exactly the same to the reference.

- For 4, the functionalities in the summary are similar to the reference. It may be vague in details, but the overall functionality and purpose is correct.

- For 3, the summary does not specify functionality. It only repeats some low-level operations without high level abstractions.

- For 2, the summary specify relevant but inaccurate functionality. The functionality specified in the summary may be relevant to the reference summary, but they have significant differences.

- For 1, the summary contains irrelevant functionality. It is contains a totally different behavior with the reference.

Your output should first briefly comment the summary from the aforementioned perspectives. Do not allow the length of the responses to influence your evaluation. Be as objective as possible.

Figure 7: Prompts for GPT4-Evaluator for asking functionality.

Table 4: Goals in reverse engineering. We construct the table from a thorough study for human reverse engineers [9](Table 12).

| Scope | Goal |
|---|---|
| Related to specific analyses | (1) Understand the purpose of analysis<br>(2) Finish the analysis quickly |
| Easy to access | (3) Discover general properties of the program<br>(e.g., size of the program) |
| Addressed by the decompiler | (4) Understand how the program uses the system interface<br>(5) Understand, abstract, and label instruction-level information<br>(6) Understand how the program uses data |
| **Can be enhanced by summarization** | (7) Construct a complete "picture" of the program<br>(8) Understand, abstract, and label the program's functions |

For other goals in Table 4, goals (1)–(2) are associated with specific analyses, instead of programs. Goal (3) aims to capture the general properties of a program (e.g., the size of a program, the sections in a binary executable file). These properties are easily accessible. The following three goals (4–6) are achieved by a decompiler. The decompiler recovers call to the system APIs (goal 4), reasons instructions and lifts them to a C-like syntax (goal 5), and recovers data dependences by introducing variables (goal 6). Therefore, the focus of binary summarization is on the last two goals, requiring understanding and reasoning of program semantics.

### B.3 Importance and Use Scenarios of Function Name Recovery

Function name recovery is important to the reverse engineering task because a human typically starts the reverse engineering task by achieving a rough understanding about all functions, as suggested by studies on human reverse engineers [9, 10]. For example, a malware sample to analyze may contain hundreds of binary functions. A reverse engineer will need to first locate functions with suspicious behaviors (e.g., executing commands received from a remote server) before analyzing the function in detail. The workload would be huge even if all functions have natural language summaries. On the other hand, if all the decompiled functions have names as in the source code, a human developer can efficiently go through the list of function names and identify functions requiring further inspection.

### B.4 Formal Definition of Precision and Recall Used by the SymLM Metrics

Formally, the precision and recall are defined as follows:

$$Precision = \frac{\|T_g \cap T_p\|}{\|T_p\|} \quad Recall = \frac{\|T_g \cap T_p\|}{\|T_g\|},$$

where $T_g$ is the token set of the groundtruth name, and $T_p$ the token set of predicted name.

## C  Prompts Used

We show our prompts to generate source code summarization in Figure 8, the prompts to generate decompiled code summarization in Figure 10, and the prompts to recovery function names in Figure 10. Note that the prompts for GPT4Evaluator are discussed in the previous section, shown in Figure 6 and Figure 7.

## D  User Study for Binary Summarization

The study leverages the same prompts used in the GPT4Evaluator (details in Appendix B.2), asking users to evaluate a piece of summary in terms of context relevance and functionality. The results are shown in Table 5. We can see that both CHRF and GPT4Evaluator are consistent with human preference, with GPT4Evaluator the most consistent metric with regard to human scores. Therefore, we use CHRF and GPT4Evaluator to evaluate the quality of binary summarizations.

> **System:** You are an experienced C/C++ software developer.
> **User:** You are provided with the following function:
> {}
> First generate a brief step-by-step description of its functionality in the format:
> **Description**: ...
> Then generate a high-level summary of its functionality in the format:
> **Summary**: The function ...
> After that, generate a brief description of its general purpose in the format:
> **Purpose**: The purpose of the function is to ...

Figure 8: Prompts for source code summarization.

> **System:** You are an experienced binary reverse engineer to understand decompiled C code that lacks symbol information.
> **User (Default):**
> You are provided with the following decompiled function that is hardly human readable:
> {}
> First generate a brief step-by-step description of its functionality in the format:
> **Description**: ...
> Then try to generate a summary of it that can help human understand / inspect its original high-level source code functionality in the format:
> **Summary**: The function ...
> After that, inspect and generate a brief description of its general purpose in the format:
> **Purpose**: The purpose of the function seems to ...
> **User (Augmented):**
> You are provided with the following decompiled function that is not human readable:
> {}
> First generate a brief step-by-step description of the functionality of the decompiled code in the format:
> **Description**: ...
> Then try to generate a summary of it that can help human understand / inspect its original high-level source code functionality in the format:
> **Summary**: The function ...
> After that, consider the following source functions (if any) that are potentially relevant to this decompiled function.
> {source functions}
> Analyze whether they are relevant to the decompiled function in the format:
> **Analysis**: ...
> Finally, based on the analysis, try to inspect and generate the general purpose of the decompiled function in the format:
> **Purpose**: The purpose of the function seems to ...

Figure 9: Prompts for decompiled code summarization. User (Default) denotes directly prompting, while User (Augmented) denotes prompting with relevant source code snippets obtained by a tool.

**User (Default):**
You have decompiled a function from an executable, which currently has a generic name like `sub_xxx`. The decompiled function code is as follows:
{}
Generate a more human-understandable function name for the decompiled code to replace the original `sub_xxx` in the format:
**Function Name**: `function_name_goes_here`
**User (Augmented):**
You have decompiled a function from an executable, which currently has a generic name like `sub_xxx`. The decompiled function code is as follows:
{}
Consider the following source functions (if any) that are potentially relevant to this decompiled function. {source functions}
Analyze whether these source functions are relevant to the decompiled function in the format:
**Analysis**: ...
Then, based on the analysis, generate a more human-understandable function name for the decompiled code to replace the original `sub_xxx` in the format:
**Function Name**: `function_name_goes_here`

Figure 10: Prompts for function name recovery. User (Default) denotes directly prompting, while User (Augmented) denotes prompting with relevant source code snippets obtained by a tool.

In a pioneer study involving 120 random samples, we find that the score distribution is highly imbalanced. Only less than 5 samples have a score of 1 or 2 for both questions. To make the study more effective, the authors manually analyze 300 cases and construct a 60-sample question set with more balanced scores (based on the scores of authors). For each question, we make sure at least 3 users give their scores, and use the median scores as the results. We randomly shuffle summaries obtained with `ProRec`, a retriever, and directly prompting to prevent undesirable bias.

Table 5: Spearman correlation between human preference and auto-metrics. Columns 2–3 and 4–5 are for the context relevance questions and functionality questions, respectively. For each question, we report both the correlation and the p-value. A higher 'correlation' value and a smaller 'p-value' indicate a statistically stronger correlation.

| Metric | Context Relevance | | Functionality | |
|---|---|---|---|---|
| | Correlation | p-value | Correlation | p-value |
| METEOR | 0.51 | 5.4e-5 | 0.39 | 2.5e-3 |
| BLEU | 0.28 | 0.05 | 0.21 | 0.11 |
| ROUGE-L | 0.49 | 1.1e-4 | 0.40 | 1.9e-3 |
| CHRF | 0.56 | **4.7e-6** | 0.48 | 1.5e-4 |
| GPT4Eval. | **0.58** | 2.7e-6 | **0.58** | **2.6e-6** |

# E  Case Study

We show two specific cases to illustrate the performance and limitation of `ProRec`. In Figure 11, we show a function that initializes a kay for encryption. Without any context information, GPT-3.5 summarizes the function with generic descriptions (e.g., "maniplate and transform the input data"). On the other hand, `ProRec` generates code snippets (only one of them is shown here) related to encryption keys. Provided with these code snippets, GPT-3.5 correctly summarizes the function as "perform cryptography operations", and mentions several operations related to "key". Although the summarization does not perfectly reflect the "initialization" purpose of this function, the description is clearer and more relevant to the context (i.e., key operations).

We study a failure case of `ProRec` for the function name recovery task. The example is shown in Figure 12. The function reads data from a temperature sensor and convert the temperature from raw

20

(a) Decompiled Code

```
__int64 __fastcall sub_4022B2(__int64 a1, _QWORD
*a2){
  j = 0;
  for (i = 15; i >= 0; --i){
    *((_BYTE *)v4 + i) = *(_BYTE *)(i + a1);
    j ^= *(unsigned __int8 *)(i + a1);
  }
  v5 = j; result = v4[0]; v3 = v4[1];
  *a2 = v4[0]; a2[1] = v3;
  for (i = 1; i <= 16; ++i){
    for (j = 16; j >= 0; --j)
      *((_BYTE *)v4 + j) =
        (8 * *(v4 + j))|(*(v4 + j) >> 5);
  result = (unsigned int)i;
  v8 = i;
  for (j = 0; j <= 15; ++j){
    result = (__int64)&a2[2 * i] + j;
    *result = *(v4 + byte_405440[v8++])
                    +byte_405120[16 * i - 16 + j];
}}}
```

(b) Source Code

```
void BTSaferpInit(const unsigned char*key, symmetric_key *skey){
  // ...
  static const unsigned char mod_table[] = {/* ... */};
  y = 0;
  for (x = 15; x >= 0; x--) {
    t[x] = key[x];
    y ^= key[x];
  }
  t[16] = y; memcpy(skey->saferp.K[0], t, 16);
  for (x = 1; x < 17; x++) {
    for (y = 16; y >= 0; y--) {
      /* ROL */
      t[y] = ((t[y]<<3)|(t[y]>>5)) & 255;
    }
    z = x;
    for (y = 0; y < 16; y++) {
      skey->saferp.K[x][y] =
          (t[mod_table[z]] + safer_bias[x-1][y]) & 255;
      z++;
    }
}}}
```

(c) Probed Code

```
static void aes_key_expansion
  (unsigned char *key, unsigned char *w){
  int i, j;
  unsigned char temp[4];

  for (i = 0; i < 16; i++)
    w[i] = key[i];

  for (i = 16; i < 176; i++) {
    for (j = 0; j < 4; j++)
      temp[j] = w[i - 1 + j];
...
```

(d) Generated Summaries

**Summary w/o probed code:** The purpose of the function seems to be to manipulate and transform the input data located at memory address `a1` and store the result in the memory locations pointed to by `a2`. …

**Summary w/ probed code:** The purpose of the function seems to be to perform cryptographic operations. It likely involves key expansion, substitution, permutation, and other cryptographic operations to transform the input data into an encrypted or decrypted form. …

Figure 11: `ProRec` helps generate a summary that is more relevant to the source code context.

sensor data to human-readable temperature unit. We show two code snippets generated by `ProRec`. We can see that `ProRec` successfully recognizes the function is relevant to read and parse data from a sensor. However, it does not accurately associate it with the temperature sensor. Therefore, although the generated summary is of better quality, the recovered function name is still different from the source code.

(a) Decompiled Code

(b) Source Code

(c) Probed Code (2 Snippets)

**Summary w/ probed code:** The purpose of the function seems to be to process data based on specific conditions and input parameters, possibly related to environmental sensor reading.

(d) Generated Summary

sensor_data_processing

(e) Recovered Function Name
(different with source code)

Figure 12: `ProRec` helps generate a summary that is more relevant to the source code context. However, the recovered function name is different from source code.