Symbol Preference Aware Generative Models for Recovering Variable Names from Stripped Binary

Xiangzhe Xu[†] Zhuo Zhang Zian Su Ziyang Huang
Shiwei Feng Yapeng Ye Nan Jiang Danning Xie
Siyuan Cheng Lin Tan Xiangyu Zhang
[†]xu1415@purdue.edu
Purdue University

***Abstract*—Decompilation aims to recover the source code form of a binary executable. It has many security applications, such as malware analysis, vulnerability detection, and code hardening. A prominent challenge in decompilation is to recover variable names. We propose a novel technique that leverages the strengths of generative models while mitigating model biases and potential hallucinations. We build a prototype, GenNm, from pre-trained generative models CodeGemma-2B and CodeLlama-7B. We fine-tune GenNm on decompiled functions and mitigate model biases by incorporating symbol preference into the training pipeline. GenNm includes names from callers and callees while querying a function, providing rich contextual information within the model's input token limitation. It further leverages program analysis to validate the consistency of names produced by the generative model. Our results show that GenNm improves the state-of-the-art name recovery precision by 8.6 and 11.4 percentage points on two commonly used datasets and improves the state-of-the-art from 8.5% to 22.8% in the most challenging setup where ground-truth variable names are not seen in the training dataset.**

I. Introduction

Deployed software often has the form of binary executable. Understanding these prevalent binaries is essential for various security and safety aspects of software, including conducting security assessments of contemporary devices such as home automation systems [4], [24] and autopilot technology [62], maintaining and hardening legacy software [15], [40], [14], detecting vulnerabilities in commercial-off-the-shelf software [65], [36], and analyzing malware that threatens our daily lives [64], [56], [5]. A significant challenge, however, is presented by the fact that most these binaries are shipped without source code, making them extremely difficult to comprehend. To bridge this gap, reverse engineering techniques have emerged to recover source-code-level details. Over the past decade, techniques like disassembly [9], [48], [41], [2], [67], function boundary identification [3], type inference [37], [54], [35], [55], [68], [47], recovery of high-level abstractions [53], code structure [8], and data structures [53] have advanced significantly.

Despite these successes, a crucial step of reverse-engineering pipelines, namely recovering variable names, remains inadequately addressed. Recovering names from fully stripped binary programs entails an in-depth understanding of both *machine semantics*, concerning data-flow and control-flow, and *descriptive semantics*, reflecting how the code is understood by human developers. This duality poses a significant challenge for conventional analysis methods [12], [66] that primarily focus on machine semantics, resulting in the failure of recovering meaningful variable names.

Recent work in renaming variables benefits from advances in machine learning models [34], [16], [46]. They formulate the problem as a classification task (a.k.a., a closed-vocabulary sequence labeling task). In the training stage, a model learns a set of variable names, i.e., the vocabulary. In the inference stage, it takes as input a decompiled function, and predicts the name for each variable by picking one from the vocabulary. Although such methods have achieved good results, their generality is limited due to the following reasons. (1) A classification model can only predict names within its training vocabulary. (2) Variable name distributions are largely biased, and it is challenging to train a good classifier on biased distributions [25], [28]. (3) Existing methods process one function at a time, due to models' input size limits or model capacity, missing important contextual information.

In particular, a classification model can only select names from the training vocabulary. It cannot "invent" new names based on program contexts. Consequently, the state-of-the-art model achieves a precision of less than 10% for variables whose ground-truth names are not in the training dataset (see Section VI-D). Moreover, distributions of variable names are biased. For example, in datasets constructed from GitHub repositories [16] or Linux packages [46], more than 50% of names appear less than 2 times, whereas 0.1% of names appear over 1,000 times. A typical classification loss that maximizes the probability of selecting the ground-truth names would undesirably emphasize the frequent names. As a result, the performance of classification model degrades by 57.5% (from 31.8% to 13.5%) for names rarely present in the training dataset (see Section VI-D). Finally, most existing models used in reverse engineering have a limited input window and hence analyze individual decompiled functions independently, missing important information in the calling context.

To address the challenges, we propose GenNm as a systematic solution to recovering names from fully stripped binaries. It leverages a generative model that composes new names from tokens. To mitigate biases in the training dataset, we propose *symbol preference optimization*. It explicitly guides the generative model to select variable names relevant to program semantics rather than simply selecting frequent names regardless of program contexts. Moreover, we augment the name generation process with contextual information via program analysis and perform naming consistency validation.

Generating name, not classifying. The generative model composes variable names from tokens, and thus can better generalize to rarely present or even unseen names. Intuitively, a human developer seldom considers complex variable names as an atomic word. Instead, she understands it as a composition of several keywords. For example, a rare name `ip_hdrlen` is composed from three frequent sub-words: `ip`, `header`, and `length`. The generative nature of model thus naturally aligns with the cognitive model of a human developer.

Incorporating symbol preference to training. We formulate the implicit biases in the training dataset as misalignment between data frequencies and *symbol preference* of developers. Symbol preference denotes that a developer prefers one name over another in a specific program context. For example, in the context of network programming, a developer typically names the data sent over network as `packet` instead of `array`, although `array` may be a more frequent name in the whole dataset. We therefore propose an additional training stage named *SYMbol Preference Optimization* (SymPO) to explicitly guide our model to pick the preferred name over the sub-optimal names under given program contexts.

Augmenting contexts via program analysis. We enhance the name generation process with program analysis to leverage the information in calling contexts and make the generated names more robust. Our design is inspired by recent studies for human reverse engineers [39], [13], which highlight the practice of inspecting all functions in a breadth-first way, and then iteratively refining the understanding for individual functions. We formulate our solution to name recovery as a similarly iterative process. Our model first generates variable names for each function based on local context (i.e., the function body). Then we follow the program call graph to *propagate* predicted names of a function to its caller and callee functions, mimicking how a human developer uses global knowledge of a binary program. Moreover, to alleviate the negative impacts of hallucination of the generative model [7], [43], [10], we devise a type-inference like algorithm that checks the semantics consistency of generated names along program data-flow.

We summarize our contributions as follows:

- We propose a novel technique for name recovery from fully stripped binaries, leveraging the strengths of both generative language models and program analysis.
- To mitigate biases entailed in the long-tail distribution of variable names, we encode the symbol preference for variable names in the training pipeline, guiding the model to select names relevant to program context with higher probabilities.
- We augment the generation process with global contextual information gathered along call graph, and alleviate the impact of model hallucination by checking name consistency along program data-flow.
- We develop a prototype GenNm (Unleashing the Power of Generative Model in Recovering Variable Names from Stripped Binary). GenNm improves the name recovery accuracy over the state-of-the-art technique [46] by 8.6 and 11.4 percentage points on two commonly used datasets, respectively. On challenging cases where the ground-truth names are not seen during training, GenNm improves over the state-of-the-art technique by 168%, i.e., 8.5% versus 22.8%.

II. Motivation and Overview

We use a motivating example to illustrate the limitations of the state-of-the-art technique for renaming decompiled variables. Following this, we demonstrate our method.

A. Motivating Example

Fig. 1a shows our motivating example, which is adapted from the function `send_packet()` in an exploit for CVE-2018-4407 [18]. The function sends a TCP packet that triggers a buffer-overflow vulnerability. The code snippet in Fig. 1a illustrates the logic to initialize the IP packet header (`ip_hdr`, lines 4–7) and compute the checksum for the TCP packet (line 9).

We show the corresponding decompiled code in Fig. 1b. Each line in the decompiled code is aligned with the related source code line, and the corresponding variables are highlighted with the same colors. For variables `s` and `n` in the decompiled code, the decompiler (IDA-Pro [30] in this case) synthesizes their names based on calls to library functions and the types of the two variables. Although synthesized names may help understanding (e.g., `n` may indicate the length of some buffer), they can hardly reflect the context and are hence much less informative than the original symbol names. For example, the source code variable related to `n` is `ip_hdrlen`, which denotes the length of the IP packet header. The synthesized name `n` fails to reflect this information. Similarly, the variable name `v29` is simply a placeholder name that is not meaningful. In both cases, the variable names in the decompiled code cannot reflect similar semantics to their source code counterparts.

The goal of variable name recovery is hence to generate meaningful names for variables with placeholder names or names synthesized from library functions.

B. Challenges and Limitations of State-of-the-Art

The state-of-the-art technique VarBERT [46] leverages the Transformer model [60] to recover variable names. The model takes as input a decompiled function, and predicts a name for each variable. The problem is formulated as a classification task. A set of variable names is first collected from the training data, noted as the vocabulary. A model is trained to select a name from the vocabulary for each variable in the decompiled function. We show with the motivating example three major challenges in recovering variables from stripped binaries, and thus discuss the limitations of state-of-the-art. The predictions of VarBERT for the motivating example are shown in the second column of Fig. 2.

Challenge 1: Cannot predict names not in the vocabulary. A classification model can only select names from those seen during training (i.e., the vocabulary). It cannot compose new names based on program contexts. In Section VI-D, we will show that 16% of the variables in our test dataset have ground-truth names not seen in the training dataset [1]. As a result, VarBERT achieves only 8.5% precision on those variables. In our motivating example, the ground-truth name for variable `n` (at line A4 in Fig. 1b) is `ip_hdrlen`, which indicates

[1]Our dataset is derived from a high-quality dataset VarCorpus, with a train-test split ratio of 9:1. See Section V for details.

```
0 int send_packet(...) {
  ...
1 memset(packet, 0, sizeof(packet));
2 struct iphdr* ip_hdr =(struct iphdr*)packet;
  ...
3 // Fill in the IP Header
4 memset(ip_hdr, 0, 0x3c);
5 ip_hdr->ihl = ip_hdrlen >> 2;
6 ip_hdr->tos = 0;
7 ip_hdr->id = htonl(...);
  ...
8 // Compute checksums
9 tcp_checksum(ip_hdr,ip_hdrlen,...);
  ...}
```

(a) Source code

```
A0int sub_401430(...){
   ...
A1 memset(s, 0, 0x400);
A2 v29 = s;
   ...

A3 memset(v29, 0, 0x3c);
A4 *v29 = (n >> 2) & 0xF;
A5 *((char*)v29+1) = 0;
A6 *((uint16*)v29+2) = htonl(...);
   ...

A8 sub_40197A(v29, n,...);
   ...}
```

(b) Decompiled code

```
int sub_401430(...){
 ...
 memset(s, 0, 0x400);
 v29 = s;
 ...
 memset(v29, 0, 0x3c);
 *v29 = (n >> 2) & 0xF;
 *((char*)v29+1) = 0;
 *((uint16*)v29+2)
               = htonl(...);
 sub_40197A(v29, n,...);
 ...}

Q: v29 , s , n
```

```
A: v29 -> ip_hdr
   s -> packet
   n -> ip_hdrlen
```

(c) Input to fine-tune GenNm

Fig. 1: Code snippets for the motivating example. Corresponding variables are highlighted with same colors.

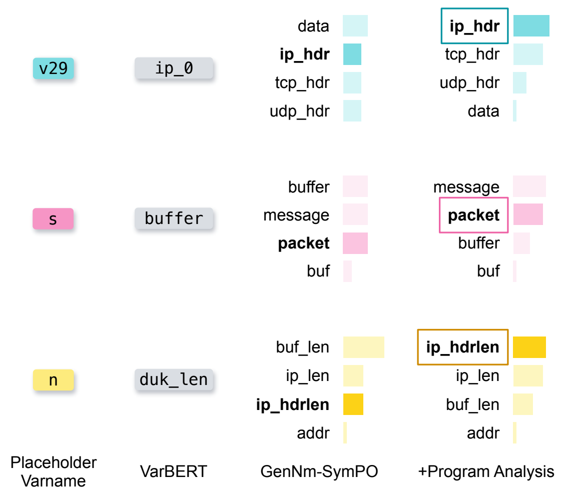


Fig. 2: Name selections for baseline (VarBERT) and name distributions for the predictions of GenNm. Each column denotes the predictions of a technique. VarBERT denotes the baseline model, *GenNm-SymPO* denotes the GenNm model after fine-tuning and symbol preference optimization. *+Program Analysis* denotes the model is used with the context information augmented by program analysis. Blue, pink, and yellow colors denote predictions for v29, s, and n. Names are ranked by their probability where a longer bar denotes a higher probability. Names highlighted with **bold fonts** are names similar or equal to ground-truth names. Names with *outlines* are those selected by the name validation algorithm.

the length of an IP packet header. However, the name never occurs in the training dataset. Thus, VarBERT mistakenly predicts the name of n as duk_len, where duk is an irrelevant program in the training dataset. For unseen names, GenNm outperforms VarBERT by 168%, i.e., 8.5% versus 22.8% (details in Section VI-D).

Challenge 2: Long-tail distribution of variable names makes correct prediction difficult. The distribution of variable names is imbalanced and has a long-tail. For example, Fig. 3 shows the distribution of names in our dataset in terms of frequency. Observe that the most frequent name appears around 50k times, while 50% of the names appear only once. It is hence challenging to train a classification model from such data with a significantly biased distribution [25],

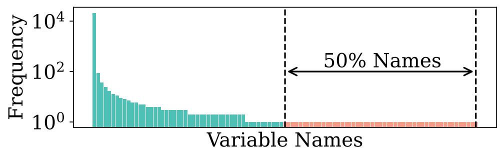


Fig. 3: Distribution of name frequencies. More than 50% variable names (in orange) appear only once in the training dataset.

[28]. A typical classification loss used in training optimizes the model's probability to predict the ground-truth name for each variable. This training loss undesirably emphasizes the frequent names. For example, the VarBERT model predicts the name of s (at lines A1–A2 in Fig. 1b) as buffer. We analyze the training dataset and find that the variable name buffer is passed as the first argument to memset for more than 500 times in the training samples. On the other hand, the ground-truth name in this case, packet, appears with memset for only 25 times in the training data. Therefore, the model biases towards the name buffer after seeing the variable is used as the first argument to memset in the query.

Challenge 3: Missing contextual information makes prediction difficult. Limited by the input length and the understanding capability of typical classification models (which are smaller than pre-trained generative models), VarBERT and many other existing works [34], [16] analyzes only one function at a time. This practice, however, misses important information from the calling context. For example, at line A8 of Fig. 1b, a model has no knowledge about the callee function sub_40197A without contextual information. Consequently, it can hardly deduce the semantics of variable v29, which is passed as the first argument to sub_40197A. VarBERT mistakenly predicts v29 as ip_0, while the ground-truth name is ip_hdr.

C. Our Method

We alleviate the closed vocabulary problem by fine-tuning generative models that can compose unseen names. To reduce the biases in the training dataset, we additionally train our models with *symbol preference optimization* that aligns the model with the symbol preference of developers. We leverage contextual information in two folds: (1) To augment a query function with a better *global* context, we propagate information of individual functions through the call graph. (2) We follow the program data-flow to impose constraints on candidate names of a variable based on other correlated variables within the function (i.e., the *local* context).

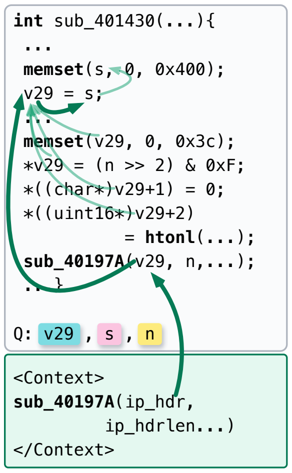
Solution for Challenge 1: Fine-tuning generative models. A generative model can concatenate multiple tokens to construct a variable name and hence has potential advantages over classification based methods. However, prior efforts [34] that trained an RNN-based generative model from scratch under-performed the SOTA classification based method VarBERT. We believe it is because the generative model is not pre-trained on a large code corpus. Hence, GenNm fine-tunes from pre-trained generative code language models, benefiting from the substantial pre-training efforts.

Large language models (LLMs) (e.g., ChatGPT, and GPT-4 [45], [43]) are advanced pre-trained generative models. They demonstrate strong capabilities in understanding both natural language text and source code. However, the distribution of decompiled code is dissimilar to either. For example, although the decompiled code by IDA has the same syntax as the C language, it is like a direct translation of individual low-level instructions instead of their abstraction. In particular, assume a source statement is compiled to ten instructions. The instructions are likely decompiled to many C statements reflecting low-level operations, instead of one natural source code statement. Such low-level C code is hence out of the training distribution of most LLMs. Our evaluation in Section VI-C shows that ChatGPT and GPT-4 under-performs our model by 11.3 percentage points in terms of precision.

To bridge the gap between the distribution of the pre-training knowledge in a generative code language model and the distribution of decompiled code, we fine-tune a generative model using decompiled code. An example input used in the fine-tuning stage is shown in Fig. 1c, where the grey box contains the query decompiled function and a list of placeholder variable names; the blue box contains the expected response of GenNm, consisting of a map from a placeholder name to a ground-truth variable name. Intuitively, the fine-tuning guides the model to generate the expected variable names based on the query function. The last row at the third column (GenNm-SymPO) of Fig. 2 shows that after fine-tuning, GenNm composes the unseen name `ip_hdrlen` as a top candidate.

Solution for Challenge 2: Symbol preference optimization (SymPO). Similar to a classification model trained only on ground-truth names, a generative model trained only on ground-truth names inherits the biases in the training dataset. Our key insight is that developers' preference over symbol names are implied by the ground-truth names, and the preference can be used to mitigate the biases in the training dataset. We propose the concept *symbol preference*, denoting that a name is preferred over other names given certain program context. For example, the variable marked in pink in Fig. 2 has the ground-truth name `packet`. That is because `packet` is more relevant to the context of network programming, and is thus more preferable than the highly frequent name `buffer`.

Technically, after training a generative model with the ground-truth names, we use the trained model to inference on the *training* dataset. We then collect the cases that the model makes mistakes. Intuitively, these counterexamples reflect the misalignment between the model's biases and the symbol preference. We adapt a loss function used in the *direct preference optimization* (DPO) [50] algorithm, guiding the model to select the preferred names over the biased ones. As a result, as shown in the third column (GenNm-SymPO) of Fig. 2, after SymPO the preferred name `packet` (in pink rows) and `ip_hdr` (in blue rows) have high probabilities, comparable to the most frequent names `buffer` (in pink rows) and `data` (in blue rows).


```
int sub_401430(...){
 ...
 memset(s, 0, 0x400);
 v29 = s;
 ...
 memset(v29, 0, 0x3c);
 *v29 = (n >> 2) & 0xF;
 *((char*)v29+1) = 0;
 *((uint16*)v29+2)
              = htonl(...);
 sub_40197A(v29, n,...);
 ... }

Q: v29 , s , n
```
```
<Context>
sub_40197A(ip_hdr,
           ip_hdrlen...)
</Context>
```
Fig. 4: Query prompt to GenNm augmented with the information propagated from the calling context (the green box). Dataflow used in name validation are indicated by green arrows, with most relevant ones highlighted.

Solution for Challenge 3: Augmenting contextual information via program analysis. Individual decompiled functions have limited contextual information. The local information in a function may not be sufficient for a model to generate correct names. Take `v29` as an example. GenNm generates with high probabilities three similar names: `ip_hdr`, `tcp_hdr`, and `udp_hdr`, as shown at the blue rows of the column GenNm-SymPO in Fig. 2. However, without the contextual information from the callee function `sub_40197A`, it is challenging to decide the precise name for `v29`. A straightforward solution to leverage global contextual information would be including caller and callee code bodies into the query. However, this naive solution incurs a substantially higher cost due to the much larger number of tokens entailed. Moreover, although LLMs have a relatively long context-window length, the performance degrades when the input becomes longer [27]. Therefore, we use function signatures as summaries for calling contexts. Specifically, we design an iterative inference process. We first ask GenNm to generate names based on local information (e.g., the function shown in the grey box of Fig. 1c) for individual functions, and then gather the predicted names along the program call graph, adding contextual information to the queries of individual functions. For example, the green box in Fig. 4 shows the context propagated to our motivating example. Note that names `ip_hdr` and `ip_hdrlen` in the green box are predicted based on the function body of the callee function `sub_40197A` (not shown in the figure). The last column (+Program Analaysis) of Fig. 2 shows the output distribution of GenNm when contextual information is introduced to the query. We can see that the model correctly predicts `v29` and `n` with the ground-truth names.

Due to the inherent uncertainty of generative models [26]

$$\begin{aligned}
&\mathcal{B} \in \texttt{Binary} ::= \{\textbf{\textit{bid}} : id,\ \textbf{\textit{funcs}} : list\ \mathcal{F}, \textbf{\textit{cg:}}set\ \mathcal{F} \times \mathcal{F}\}\\
&\mathcal{F} \in \texttt{Function} ::= \{\textbf{\textit{fid}} : id,\ \textbf{\textit{body}} : str, \textbf{\textit{ids}} : set\ id\}\\
&\mathcal{N} \in \texttt{NameMap} ::= id \to id \to str\\
&\mathcal{D} \in \texttt{Dataset} ::= list\ (\ \mathcal{B} \times \mathcal{N}\ )
\end{aligned}$$

$$\begin{aligned}
&\langle \textit{FuncBody} \rangle\ \mathcal{F}.\textbf{\textit{body}} ::= \mathcal{R}; \mathcal{S} \quad \langle \textit{Params} \rangle\ \mathcal{R} ::= list\ id\\
&\langle \textit{Expr} \rangle\ \mathcal{E} ::= id \mid id(\mathcal{A}) \mid Other \quad \langle \textit{Args} \rangle\ \mathcal{A} ::= list\ \mathcal{E}\\
&\langle \textit{Stmt} \rangle\ \mathcal{S} ::= \mathcal{S}_1; \mathcal{S}_2 \mid \mathcal{E}_0 := \mathcal{E}_1 \mid \textbf{return}\ \mathcal{E} \mid \textbf{while}(\mathcal{E})\{\mathcal{S}\} \mid\\
&\qquad \textbf{if}(\mathcal{E})\{\mathcal{S}_1\}\textbf{else}\{\mathcal{S}_2\}
\end{aligned}$$

Fig. 5: Definitions

and the challenges of understanding programs without symbols [58], we observe that a generative model may miss the information in the local context, predicting inaccurate names even if the correct global context is provided. For example, in the fourth column (+Program Analysis) of Fig. 2, GENNM predicts `message` for the variable `s` (at the pink rows). Therefore, we propose a name validation algorithm to select (from top-ranked candidates) the name that is most consistent with the local program context. We consider variable names as type-like properties, and use inference rules to propagate names along program data-flow. For example, to select the best name for variable `s`, the data-flow edges highlighted in Fig. 4 connects it to `v29`, and `v29` is further connected to the first argument `ip_hdr` of the callee function `sub_40197A`. They indicate the names of those variables may have semantics relevance with `s`. GENNM calculates the semantics similarity between the names of those variables and the candidate names of `s` (i.e., `message`, `packet`, `buffer`, and `buff`). It then finds that the name `packet` is the most relevant with the names of the other two variables.

III. PROBLEM DEFINITION

To facilitate discussion, we formalize the problem as shown in Fig. 5. We use id to refer to the placeholder names synthesized by the decompiler, and use $name$ to refer to meaningful names. A binary program consists of an id, a list of binary functions, and a call graph. The call graph is a set of edges from caller functions to callee functions. A decompiled function consists of a function id, the string of decompiled code, and a set of identifiers used in the function. A name map is associated with a binary program. It takes as input the id of a function, the id of a variable in this function, and returns a meaningful name for the variable. The dataset of binary programs $\mathcal{D}$ has the type of a list of pairs. Each pair consists of a binary program and the corresponding name map containing the ground-truth names.

We transform the decompiled code of a function to a program in a simple language to simplify the discussion. The language definition is shown in the lower part of Fig. 5. The definitions are standard. Note that we omit most types of expressions and only focus on expressions containing an identifier (id) and a function call ($id(\mathcal{A})$).

IV. METHOD

A. Overview

Training. We show the training pipeline of GENNM in Fig. 6. We train GENNM in three steps: (1) The training process starts from a pre-trained checkpoint of a code language model (e.g., CodeLlama-7B). It first fine-tunes the pre-trained model on decompiled code to align the distribution of the pre-trained model to the distribution of decompiled code (and the ground-truth names), resulting a model noted as GENNM$_{\text{Ctx}}$. (2) We use GENNM$_{\text{Ctx}}$ to inference on the training dataset, and construct a pairwise symbol preference dataset from model's predictions. Each data sample in the symbol preference dataset contains a preferred name and a less preferred name. (3) We further train the model with the symbol preference optimization on the preference dataset, resulting a model noted as GENNM$_{\text{SymPO}}$.

Inference. The inference process is depicted by Fig. 7. We solve the name recovery problem with an iterative process. At each round, the GENNM model predicts names for individual decompiled functions, using the global contextual information collected from previous rounds (Step 1). Then the predictions are added to a candidate name map from a variable to the candidate names of this variable seen across rounds (Step 2). We then leverage the name validation algorithm to select best candidate names based on program data-flow (Steps 3–4). Finally, the selected names are propagated following the call-graph, updating the quries to the GENNM model (Step 5) in the next round. It terminates when no variable name is updated or until a predefined budget is reached.

We discuss the training pipeline in Sections IV-B and IV-C. The inference process is discussed in Section IV-D.

B. Fine-tuning Generative Model

To bridge the gap between the distribution of a pre-trained code language model and the decompiled code, we fine-tune our model from checkpoints of a pre-trained model (e.g., CodeLlama-7B). Our fine-tuning involves two types of datasets: one dataset that contains individual decompiled functions and the corresponding ground-truth variable names and the other dataset that additionally contains the global contextual information obtained following the program call graph. We fine-tune a model with both datasets because we want our model to have the capabilities of inferring names from local information and generating names considering global contextual information. The training objective aligns with how the fine-tuned model is used in the inference stage. We leverage the *causal language modeling* (CLM) [49] loss for fine-tuning. The loss is computed on tokens in both the query decompiled functions and the output names.

Dataset w/ local information. We note the dataset that contains individual decompiled functions as D_{loc}. Formally, the dataset D_{loc} is defined as follows:

$$\begin{aligned} D_{\text{loc}} ::= \big\{ &(\textbf{\textit{query}} : (f.\textbf{\textit{body}}, f.\textbf{\textit{ids}}),\\ &\textbf{\textit{resp}} : n[f.\textbf{\textit{fid}}]) \big| (b, n) \in \mathcal{D} \wedge f \in b \big\}, \end{aligned} \tag{1}$$

where $\mathcal{D}$ denotes the list of binary programs used for training, and b and n a binary and its name map, respectively, as defined in Fig. 5. Hence $n[f.\textbf{\textit{fid}}]$ denotes the map from a placeholder variable name to the ground-truth variable name for function f.

Context Propagation. We first discuss the context propagation algorithm that gathers names following the program call graph, and then discuss how we use it to construct the dataset with additional contextual information. Note that the algorithm is used to construct contextual dataset during the training time

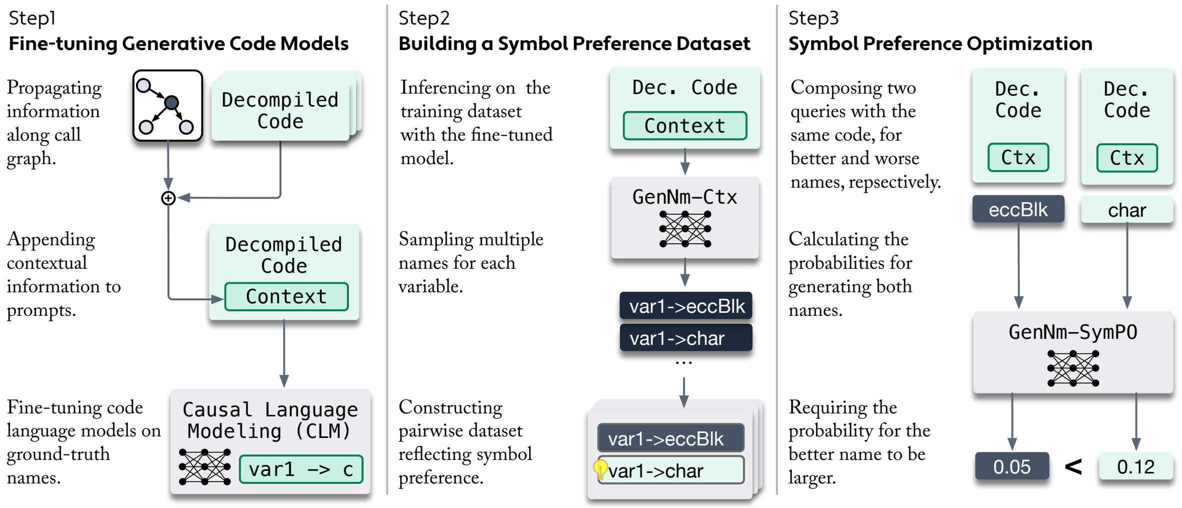

Step1
Fine-tuning Generative Code Models
Propagating information along call graph.
Decompiled Code
Appending contextual information to prompts.
Decompiled Code
Context
Fine-tuning code language models on ground-truth names.
Causal Language Modeling (CLM)
var1 -> c
Step2
Building a Symbol Preference Dataset
Inferencing on the training dataset with the fine-tuned model.
Dec. Code
Context
GenNm-Ctx
Sampling multiple names for each variable.
var1->eccBlk
var1->char
Constructing pairwise dataset reflecting symbol preference.
var1->eccBlk
var1->char
Step3
Symbol Preference Optimization
Composing two queries with the same code, for better and worse names, repsectively.
Dec. Code
Ctx
Dec. Code
Ctx
eccBlk
char
Calculating the probabilities for generating both names.
GenNm-SymPO
Requiring the probability for the better name to be larger.
0.05
<
0.12

Fig. 6: Training pipeline of GenNm

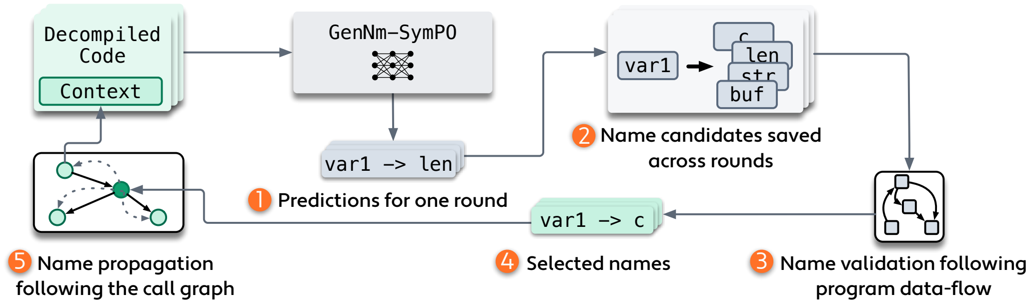

Decompiled Code
Context
GenNm-SymPO
var1 -> len
1 Predictions for one round
var1
c
len
str
buf
2 Name candidates saved across rounds
3 Name validation following program data-flow
var1 -> c
4 Selected names
5 Name propagation following the call graph

Fig. 7: Inference pipeline of GenNm

and to propagate and update model query inputs during the inference time.

The context propagation algorithm takes as input the call graph of a program and the predictions for individual functions and propagates the predicted names along the call graph. Intuitively, the propagation algorithm gathers information from both the caller functions and the callee functions of an analyzed function f. For the caller functions, the algorithm identifies the callsites, i.e., the call expressions that call f. It then renames the placeholder names in the corresponding call expressions with the names predicted from the local context of the caller function, and appends the renamed call expressions to the query of f. Similarly, the algorithm renames the signature of the callee functions of f and appends them to the query of f.

Given a function f, we formally define the context propagation rules as follows:

$$\begin{aligned} \mathit{CallerCtx}(f,n) ::= & \bigcup \{\textbf{rename}(f.\textbf{\textit{fid}}(a), n[\mathit{clr}.\textbf{\textit{fid}}]) \\ & \mid (\mathit{clr}, f) \in b.\textbf{\textit{cg}} \wedge f.\textbf{\textit{fid}}(a) \in \mathit{clr}.\textbf{\textit{body}}\} \end{aligned} \quad (2)$$

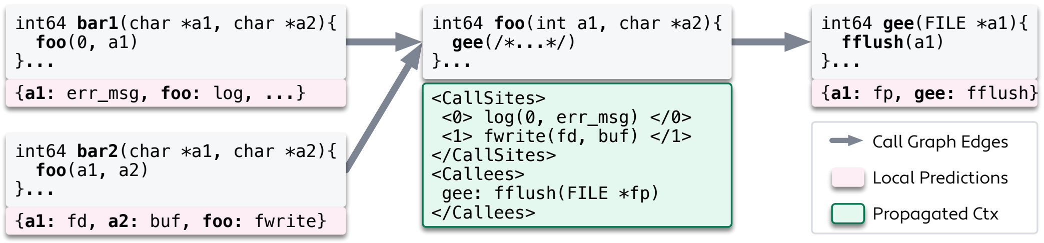
$$\begin{aligned} \mathit{CalleeCtx}(f,n) ::= & \bigcup \{\textbf{rename}(\mathit{cle}.\textbf{\textit{fid}}(r), n[\mathit{cle}.\textbf{\textit{fid}}]) \\ & \mid (f, \mathit{cle}) \in b.\textbf{\textit{cg}} \wedge (r, \_) = \mathit{cle}.\textbf{\textit{body}}\} \end{aligned} \quad (3)$$

$$\mathit{Ctx}(f,n) ::= \mathit{CallerCtx}(f,n) \cup \mathit{CalleeCtx}(f,n) \quad (4)$$

where b and n are the binary program that the function f belongs to and the corresponding name map. The name map contains the ground-truth names when constructing the training dataset and the predicted names when propagating names during inference. The utility function **rename**(x, y) renames all ids in x according to the name map y.

Given a data sample containing a decompiled function f, Equation 2 depicts the rule to propagate names from its caller. Specifically, $(clr, f) \in b.\textbf{\textit{cg}}$ describes the constraint that clr is a caller of f, and $f.\textbf{\textit{fid}}(a)$ refers to a call expression in the body of clr that calls f; $f.\textbf{\textit{fid}}$ denotes the placeholder name of f and a denotes the argument list. The propagation algorithm uses names in $n[clr.\textbf{\textit{fid}}]$ to rename the placeholder names in the call expression, and then adds it to the context of f. Similarly, Equation 3 depicts the rule to propagate context from the callee of f. As defined in Fig. 5, r denotes the parameter list of cle, and $cle.\textbf{\textit{fid}}$ refers to the placeholder name of cle. Therefore, $cle.\textbf{\textit{fid}}(r)$ denotes the signature of the callee function cle. The algorithm renames the placeholder names in the signature of the callee function and adds it to the context of f.

Fig. 8 shows a concrete example. Assume the analyzed function is `foo()`. We show two caller functions `bar1()` and `bar2()` (first column) and a callee function `gee()` (third column). For each function, we use a pink box to show the predictions obtained by analyzing the function independently. The green box under `foo()` shows the propagated contextual information. For example, in `bar1()`, the model predicts the names `err_msg` and `log` for `a1` and `foo`, respectively. Therefore, in the context of `foo()` in the middle column, the algorithm renames the call statement to `foo()` with the predicted names and propagates it as the 0-th entry of the callsites. Similarly, it renames the signature of the callee



```
int64 bar1(char *a1, char *a2){
  foo(0, a1)
}...
{a1: err_msg, foo: log, ...}

int64 bar2(char *a1, char *a2){
  foo(a1, a2)
}...
{a1: fd, a2: buf, foo: fwrite}

int64 foo(int a1, char *a2){
  gee(/*...*/)
}...
<CallSites>
 <0> log(0, err_msg) </0>
 <1> fwrite(fd, buf) </1>
</CallSites>
<Callees>
 gee: fflush(FILE *fp)
</Callees>

int64 gee(FILE *a1){
  fflush(a1)
}...
{a1: fp, gee: fflush}
```

Fig. 8: Example of propagating global contextual information along the call graph

function `gee()` with the predicted names, and propagates the renamed signature to the analyzed function.

An alternative design is simply appending the bodies of caller and callee functions to a query function. As discussed in Section II-C, it is neither efficient since it significantly increases the number of query tokens nor effective due to the degradation of model's performance with longer input context.

Dataset w/ contextual information. We formally define the dataset with contextual information (noted as D_{ctx}) as follows:

$$D_{\text{ctx}} ::= \big\{(\textit{\textbf{query}} : (f.\textit{\textbf{body}}, Ctx(f,n), f.\textit{\textbf{ids}}), \\ \textit{\textbf{resp}} : n[f.\textit{\textbf{fid}}])\big|(b,n) \in \mathcal{D} \wedge f \in b\big\}, \tag{5}$$

where $\mathcal{D}$ denotes binaries used for training, and b and n a binary and its name map, respectively, as defined in Fig. 5; $Ctx(f,n)$ denotes the contextual information gathered by the context propagation algorithm.

Loss function for fine-tuning. We use a CLM loss to fine-tune on both datasets. The loss is formally defined as follows:

$$\mathcal{L}_{\text{ft}}(\Theta, D_{\text{loc}}, D_{\text{ctx}}) ::= \\ -\mathbb{E}_{(q,r)\sim D_{\text{loc}}\cup D_{\text{ctx}}}\Big[\sum_{i=1}^{\text{len}(q)+\text{len}(r)} \log P(\mathbf{x}_i|\mathbf{x}_{<i};\Theta)\Big], \tag{6}$$

where Θ denotes the weights of the fine-tuned model; $\mathbf{x}$ denotes the sequence obtained by concatenating the tokens in the query (q) and the tokens in the response (r); $\mathbf{x}_i$ denotes the i-th token in $\mathbf{x}$; and $\mathbf{x}_{<i}$ the token sequence before the i-th token. Note that our fine-tuning stage calculates the CLM loss for tokens in both the query and the response to help the model understand the distribution of the decompiled code in the query.

C. Symbol Preference Optimization

In the natural language domain, *preference* denotes that a natural language sentence output by a generative model is preferred over another. Preference optimization is a method to align the behavior of a pre-trained LLM to human preference [50]. It takes as input pairwise data samples, and asks a model to predict a higher probability for the preferred response and a lower probability for the less preferred response. Since our technique is based on generative models, in order to counter biases, we design a SymPO method for our task. The SymPO dataset contains pairwise data samples. Each sample consists of a query function, a less preferred name (indicating the model's biases), and a preferred name. Both are sampled from the model's output. Instead of involving a human evaluator, we use the string similarity to the ground-truth name as the preference for a given variable name. The SymPO loss is carefully designed so that it teaches the model to select preferred names over the less preferred names while not compromising the model's capability on the variable recovery problem.

We first introduce how we construct the pairwise dataset used for SymPO (i.e., Step 2 in Fig. 6), and then introduce the SymPO loss (i.e., Step 3 in Fig. 6).

Constructing the SymPO dataset. We construct the dataset using GenNm$_{\text{Ctx}}$ to inference a subset of the training data, and sampling the top 20 predictions from the model for each query. We collect cases where GenNm$_{\text{Ctx}}$ makes mistakes but has at least another response in the top 20 predictions that is significantly better. Intuitively, the model has the knowledge of better names for those cases, yet it makes mistakes due to the biases. The SymPO process thus has the chance to fix the biases without changing the model significantly.

An alternative design is to use the ground-truth as the preferred names. However, the results in Appendix D show that using ground-truth names underperforms compared to using the best predictions of GenNm$_{\text{Ctx}}$ as the preferred names. We speculate that is because GenNm$_{\text{Ctx}}$ may not learn how to generate the ground-truth names for certain programs. Cases where the ground-truth diverge too much from GenNm$_{\text{Ctx}}$'s learned distribution negatively affect the model's performance.

We formally present the SymPO dataset as follows. First, we use $\hat{D}$ to denote the inferenced training subset.

$$\hat{D} ::= \big\{(\textit{\textbf{query}} : q, \textit{\textbf{preds}} : \hat{r}, \textit{\textbf{gt}} : r)\big|(q,r) \in \mathcal{D}_{\text{loc}} \cup \mathcal{D}_{\text{ctx}} \\ \wedge \hat{r} = \text{GenNm}_{\text{ctx}}(q, \text{top20})\big\}, \tag{7}$$

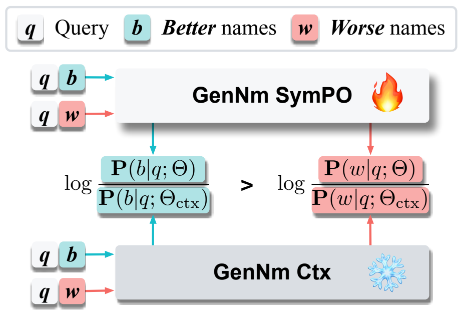
where $\text{GenNm}_{\text{ctx}}(q, \text{top20})$ denotes the top 20 responses returned by GenNm$_{\text{Ctx}}$ given a query q.

The SymPO dataset, noted as D_{prf}, is defined as follows:

$$D_{\text{prf}} ::= \big\{(\textit{\textbf{query}} : q, \textit{\textbf{better}} : r_b, \textit{\textbf{worse}} : r_w) \\ \big|(q,\hat{r},r) \in \hat{\mathcal{D}} \wedge r_w = \textbf{sample}(\hat{r}) \\ \wedge r_b = \textbf{best}(\hat{r}, r) \wedge r_b \succ_r r_w\big\}, \tag{8}$$

where $\textbf{best}(\hat{r}, r)$ denotes the name in $\hat{r}$ that is most similar to the given ground-truth name r, $\textbf{sample}(\hat{r})$ denotes a name that is randomly sampled from $\hat{r}$, and $r_b \succ_r r_w$ denotes that the name r_b is significantly more similar to the given ground-truth name r than r_w.

Moreover, to reduce the noise in the SymPO dataset and improve the training efficiency, we use lightweight static code features as heuristics to filter out low-quality data. Empirically, our static heuristics reduce the dataset size by 60%, and results in Section VI-E show that the performance achieved by training on the reduced dataset is even slightly better than training on all the data samples. In particular, we remove the functions whose local information is not enough for the model



q Query b Better names w Worse names
q b
q w
GenNm SymPO
$\log \frac{\mathbf{P}(b|q;\Theta)}{\mathbf{P}(b|q;\Theta_{\text{ctx}})} > \log \frac{\mathbf{P}(w|q;\Theta)}{\mathbf{P}(w|q;\Theta_{\text{ctx}})}$
q b
q w
GenNm Ctx

Fig. 9: Loss for SymPO. The weights of GENNM$_{\text{Ctx}}$ are frozen. The weights of GENNM$_{\text{SymPO}}$ are optimized guided by the SymPO loss. Preferred (better) names and the corresponding probabilities are in blue; less preferred (worse) names and the probabilities are in red.

to predict good names. Optimizing model's preference on those data samples introduces only noises and is not meaningful. In addition, we remove functions with less than 5 statements and meanwhile do not have branches.

Loss function of SymPO. The loss function of SymPO is adapted from the loss function proposed in direct preference optimization [50], which is used to align human preference with fine-tuned LLMs. The loss function has two subgoals: (1) guiding the model to generate better names with higher probabilities (than the probabilities for generating worse names), and (2) preventing the model from diverging too much from its original distribution. The loss function is formally presented as follows:

$$\mathcal{L}_{\text{SymPO}}(\Theta, \Theta_{\text{ctx}}) ::= \\ -\mathbb{E}_{(q,b,w)\sim D_{\text{prf}}}\left[\log\sigma\left(\beta\log\frac{\mathbf{P}(b|q;\Theta)}{\mathbf{P}(b|q;\Theta_{\text{ctx}})} - \beta\log\frac{\mathbf{P}(w|q;\Theta)}{\mathbf{P}(w|q;\Theta_{\text{ctx}})}\right)\right] \quad (9)$$

where Θ and Θ_{Ctx} denotes the weights of GENNM$_{\text{SymPO}}$ and the weights of GENNM$_{\text{Ctx}}$, respectively. β is a hyper-parameter that controls the sensitivity of the loss to the margin between the probability for better names and the probability for worse names. The loss is optimized w.r.t. Θ only. In other words, the weights of GENNM$_{\text{Ctx}}$ are frozen during SymPO.

An intuitive explanation for the loss function is visualized in Fig. 9. Two models are involved in SymPO. The first model is GENNM$_{\text{SymPO}}$, which is optimized by the loss function. It is initialized with the weights of GENNM$_{\text{Ctx}}$. The other model is a frozen GENNM$_{\text{Ctx}}$ model. Its weights will not be updated during training. It is used as a "reference" model so that the divergence of GENNM$_{\text{SymPO}}$ (from the reference model) is constrained. A detailed discussion for the rationale of using a reference model is in Appendix A. Assume a data sample consisting of the query function (q), a better name (b), and a worse name (w). The blue parts in Fig. 9 depict the first loss term in Equation 9 (i.e., $\log\frac{\mathbf{P}(b|q;\Theta)}{\mathbf{P}(b|q;\Theta_{\text{ctx}})}$). It uses both models to calculate the probabilities of generating the better name b, respectively. It guides GENNM$_{\text{SymPO}}$ to produce a significantly larger probability for b compared to GENNM$_{\text{Ctx}}$. Similarly, the red parts (corresponding to the second loss term) requires GENNM$_{\text{SymPO}}$ to generate a significantly smaller probability compared to the GENNM$_{\text{Ctx}}$.

Algorithm 1: Name Validation

```
Input: $B$: a binary program
Input: $N_0$: $id \to id \to str$, a map from a variable to an initially selected name
Input: $\hat{N}$: $id \to id \to list\ str$, a map of name candidates
Output: $N$: $id \to id \to str$, a map from a variable to the name selected by the algorithm
1 $update \leftarrow True$
2 $N_{current} \leftarrow N_0$
3 while $update$ do
4     $N_{prev} \leftarrow N_{current}$
5     $\hat{N}' \leftarrow correlated\_names(B, N_{current})$
6     $N_{current} \leftarrow semantics\_vote(\hat{N}, \hat{N}')$
7     $update \leftarrow N_{prev} \neq N_{current}$
8 return $N_{current}$
```

D. Context Augmentation via Program Analysis

At the inference stage, we iteratively run GENNM because the input contexts provided to the models are updated based on the latest round of predictions. In each iteration, the newly generated names along with the names generated in the previous rounds are considered candidate names for the variable. We discuss the context propagation algorithm (Step 5 in Fig. 7) in previous sections. Therefore, this section mainly discusses the name validation algorithm (Step 3 in Fig. 7) that leverages program analysis to aggregate the names predicted in different rounds under individual contexts. The insight is that names correlated through data flow ought to have a certain level of consistency (in terms of their natural language semantics), although they may not be identical. For example, a variable named `fout` may be passed as an argument to a parameter named `stream`, but is less likely to be assigned to a variable named `size`.

The name validation process in GENNM first extracts correlated name candidates for a variable and then selects the candidate with the maximum level of consistency. That is, our goal is to achieve a minimal total semantics distance for all correlated names. We formally define the name validation process in Algorithm 1. The algorithm takes as input a binary program, an initial name map from a variable to its initially selected name, and a map from each variable to a list of its candidate names. It outputs an updated name map from each variable to its (updated) name. At the beginning of the name validation process, the initial name map is constructed as a map from a variable to the top-1 predicted name of the generative model. The name validation algorithm consists of a loop. In each iteration, it collects additional names for each variable by inheriting names from other variables that have *direct* data flow with the variable (line 5 and details explained later), selects the best name of a variable by *semantics voting* (line 6), which will be explained later in the section, and updates the name map. It may take multiple iterations to propagate names to places that are multiple data-flow edges away. The algorithm terminates when no variable is updated or until a predefined budget is reached.

Collecting correlated name by data-flow. If a variable is directly copied to another variable, called *a direct use of the variable*, their names should be semantically consistent, analogous to type consistency. In this step, we propagate names across such direct uses to populate the candidate set and enable inconsistency suppression. Note that such propagation is not performed for composite operations. For example, the name of a right-hand-side variable involved in a binary operation may not be semantically consistent with the name of either left-hand-side variable.

We model the correlation extraction process as finding solutions to program constraints. The key constraint rules are shown in Fig. 10. The other rules are in Fig. 15 in the Appendix. The analysis takes as input a binary program and the corresponding name map. Its outputs are a correlated variable map π that maps from a variable (referred by a function id and a variable id) to a list of correlated names. For each variable, the auxiliary data structure σ maintains the origin of each correlated name to prevent duplication. Note that if an origin variable has many data-flow paths to another variable, its name may get propagated multiple times and have an in-appropriate weight in the later voting step. Specifically, an element in σ is a triple, noted as $(fid, vid, name)$, where fid and vid refers to the origin variable of a correlated name $name$.

A rule $\frac{A \; B}{C}$ is interpreted as follow: when A and B are satisfied, C is satisfied. The notation $env \vdash Stmt : env'$ is interpreted as given an environment env, the environment env' satisfies the constraints introduced by $Stmt$. The two special rules **Init** and **Out** are evaluated only once at the beginning and ending of the analysis, respectively. **Init** initializes the correlated name set of a variable to its initially selected name. The rule **Out** converts the correlated name set to the correlated name list for all variables.

The rule **Assign** depicts the constraint introduced by an assignment statement from id_1 to id_0. The rule requires all the correlated names of id_1 to be in the correlated name set of id_0 as well. We assume the name of (the destination variable) id_0 to be more general than the name of (the source variable) id_1. Therefore, the names correlated to id_1 should also have correlation with id_0. For example, assume an assignment statement `ptr=msg`. The name `ptr` is more general than the name `msg`. Thus a correlated name of `msg` (e.g., `buffer`) is likely to be a correlated name of `ptr` as well. Moreover, as depicted by the rule **Assign-R**, we propagate the selected name of id_0 to id_1 because the denotation of id_0 and id_1 may be similar. However, we do not propagate the correlated names of id_0 to id_1 because not all the names correlated to id_0 are necessarily correlated to id_1, assuming the name of id_0 denotes a broader range of semantics than the name of id_1.

The intuition of rules **Call** and **Ret** are the same. A function call would have implicit assignments between the arguments and the parameters. And the function return would have an implicit assignment between the return value and the variable storing calling results in the caller function. The dual rules **Call-R** and **Ret-R** are in Fig. 15 in the Appendix for simplicity.

Semantics voting. Semantics voting takes as input the candidate name map $\hat{N}$ (produced by the model) and the correlated name map $\hat{N}'$. For each variable, it picks the candidate name that is most similar to all correlated names and other candidate names. It is hard to directly compare the semantics similarity of two strings. Therefore, our algorithm encodes all correlated names and all candidates names to their embeddings, and measures the similarity between two names by calculating the cosine similarity. Formally, the semantics voting process for a given variable is shown as follows:

$$Candi := \hat{N}[fid][vid] \quad Corr := \hat{N}'[fid][vid]$$
$$\forall fid\ vid, \operatorname*{argmax}_{n \in Candi} \sum_{m \in [Corr; Candi]} \langle \mathbf{e}_n, \mathbf{e}_m \rangle$$

where $\langle \cdot, \cdot \rangle$ denotes cosine similarity between two embeddings and $[\cdot;\cdot]$ denotes list concatenation.

V. Experimental Setup

A. Dataset

We evaluate GenNm on two commonly used [46], [34], [16] datasets. The first one is built following the same process of DIRTY [16] (noted as the DIRTY dataset) and the other is derived from the released VarCorpus dataset used by VarBERT [46] (noted as the VarCorpus dataset). The DIRTY dataset is built from popular GitHub projects, and the VarCorpus dataset is built (by the VarBERT authors) from a Linux package manager Gentoo [21]. We rebuild the DIRTY dataset because the original DIRTY dataset contains binary programs that are not fully-stripped [46]. Additionally, the dataset provided by DIRTY's authors contains only preprocessed data without raw binaries. Our technique requires call graphs of programs, and thus cannot directly use the provided DIRTY dataset. For the VarCorpus dataset, thanks to the help of VarBERT's authors, we obtain the corresponding binary programs in VarCorpus and thus can reuse the processed VarCorpus dataset with the call graphs extracted from the binary programs. For both datasets, the ground-truth variable names are obtained from the debug information in binary programs. For the DIRTY dataset, we reuse the code provided by DIRTY's authors to collect the ground-truth. For the VarCorpus dataset, we directly reuse the ground-truth provided in the dataset.

Data quality. To prevent the data duplication problem as observed by the previous work [46], we ensure the high quality of both datasets with strict deduplication rules, only including a binary program if at least 70% of its functions are not seen. In the deduplication process, we conservatively consider two functions as same functions if they have the same name. The rationale are discussed in Appendix B. As a result, our processed datasets are more diversified than the existing datasets. For example, only 46% functions in the original VarCorpus dataset have unique names, indicating that the other 54% functions may have similar semantics (an example is in Fig. 16 in the Appendix). On the other hand, 81.3% and 89.4% functions in our processed VarCorpus and DIRTY datasets have unique names, respectively. Our processed DIRTY and VarCorpus datasets have 348k and 895k functions, respectively. Please see Appendix B for detailed statistics.

Preventing data leakage. Moreover, we use string-similarity-based rules to filter out the overlap between training and test data, preventing potential data leakage. Previous works [46], [16] use exact string match as the criterion for checking data leakage. However, as shown in Fig. 16 and Table IV in the Appendix, there might be potential data leakage even if the strings of two functions are not exactly the same (e.g., two functions may differ in only one number). To better measure the generalizability of models, we conservatively filter out those potential leakage by filtering out a test sample if its

Input: $B : \mathcal{B},\ N_{in} : \mathcal{N}$ **Output:** $\pi : id \to id \to \textit{list str}$
Auxiliary Data: $\sigma : id \to id \to \textit{set}\ (id \times id \times \textit{str})$ **State Configuration:** $\langle \pi, \sigma \rangle$
We use f to denote the function that the analyzed statement belongs to.

$$\frac{\sigma_0[fid][vid] = n \qquad (fid, vid, n) \in N_{in}}{\langle \cdot, \cdot \rangle \vdash: \langle \cdot, \sigma_0 \rangle}\textbf{Init} \qquad \frac{\pi[fid][vid] = \big[n | (\cdot, \cdot, n) \in \sigma[fid][vid]\big] \qquad (fid, vid) \in \sigma}{\langle \cdot, \sigma \rangle \vdash \textit{Done} : \langle \pi, \sigma \rangle}\textbf{Out}$$

$$\frac{\sigma' = \sigma\big[f.\textbf{\textit{fid}}, id_0 \rightsquigarrow \sigma[f.\textbf{\textit{fid}}][id_0] \cup \sigma[f.\textbf{\textit{fid}}][id_1]\big]}{\langle \cdot, \sigma \rangle \vdash id_0 := id_1 : \langle \cdot, \sigma' \rangle}\textbf{Assign} \qquad \frac{\begin{array}{c} n_1 = N_{in}[f.\textbf{\textit{fid}}][id_0] \\ \sigma' = \sigma\big[f.\textbf{\textit{fid}}, id_1 \rightsquigarrow \sigma[f.\textbf{\textit{fid}}][id_1] \cup \{(f.\textbf{\textit{fid}}, id_0, n_1)\}\big] \end{array}}{\langle \cdot, \sigma \rangle \vdash id_0 := id_1 : \langle \cdot, \sigma' \rangle}\textbf{Assign-R}$$

$$\frac{\begin{array}{c} f_1 \in B.\textbf{\textit{funcs}} \qquad id_0 = args[i] \qquad p_0 = r[i] \qquad r, \_ = f_1.\textbf{\textit{body}} \\ \sigma' = \sigma\big[f_1.\textbf{\textit{fid}}, p_0 \rightsquigarrow \sigma[f_1.\textbf{\textit{fid}}][p_0] \cup \sigma[f.\textbf{\textit{fid}}][id_0]\big] \end{array}}{\langle \cdot, \sigma \rangle \vdash f_1.\textbf{\textit{fid}}(args) : \langle \cdot, \sigma' \rangle}\textbf{Call} \qquad \frac{\begin{array}{c} f_1 \in B.\textbf{\textit{funcs}} \qquad id_1 := f.\textbf{\textit{fid}}(...) \in f_1.\textbf{\textit{body}} \\ \sigma' = \sigma\big[f_1.\textbf{\textit{fid}}, id_1 \rightsquigarrow \sigma[f_1.\textbf{\textit{fid}}][id_1] \cup \sigma[f.\textbf{\textit{fid}}][id_0]\big] \end{array}}{\langle \cdot, \sigma \rangle \vdash \textbf{return}\ id_0 : \langle \cdot, \sigma' \rangle}\textbf{Ret}$$

Fig. 10: Correlation Extraction Rules

string similarity score to a training sample (from 0 to 100) is higher than 90.

Data availability. We will publish all scripts and artifacts upon publication. Note that we reuse the VarCorpus dataset released/shared by VarBERT's authors, and thus constructing the VarCorpus dataset is not our contribution.

B. Splits

We split both datasets with a ratio of 9:1 by binaries (not by functions) for training and test. We randomly sample 5% functions from the training datasets as the validation sets. We split our training and test datasets by binary programs (instead of by binary functions). That is because splitting data by functions may cause data leakage. Decompilers typically use the address of a global variable or a function to construct a placeholder name for it. For example, assume two functions from a binary program, and both of them use a global variable `qword_409abc`. One of the functions is in the training dataset, and hence the training process exposes the ground-truth name, e.g., `message`, to the model. During test, the model can easily predict `qword_409abc` as `message` since it is already seen in the training data.

C. Models

For the DIRTY dataset, we train two GENNM models from two code language models with different sizes: one from CodeGemma-2B [59] and the other from CodeLlama-7B [51]. For the VarCorpus dataset, we only train GENNM from CodeGemma-2B due to limited resources. The detailed hyperparameters of our model are listed in Table V of the Appendix. We use VarBERT [46] as the baseline because it is a representative classification based method that demonstrates better performance than previous state-of-the-art models [34], [16]. We train all models until they converge (i.e., the validation loss no longer decreases). We select models that achieve the best validation loss.

D. Metrics

We use two sets of metrics to evaluate model performance.

Token-based semantics match. Previous works use exact string match to evaluate the performance of a variable name recovery technique. However, exact string match cannot faithfully reflect the capability of a tool. As discussed in SymLM [32], a previous work focusing on recovering function names, even when two variables have the same meaning, the names specified by developers may vary due to many reasons, e.g., use of abbreviations and concatenation of names. We thus adapt the same metrics used in SymLM to measure the quality of generated names. Intuitively, given a ground-truth name n and a predicted name $\hat{n}$, the metric tokenizes both names into sets of tokens, noted as W and $\hat{W}$. Then it uses set comparison to calculate precision and recall. Formally,

$$\textit{Precision}(W, \hat{W}) = \frac{\big\|\{\hat{w} | \hat{w} \in \hat{W} \wedge \exists w \in W, \hat{w} \simeq w\}\big\|}{\|\hat{W}\|} \tag{10}$$

$$\textit{Recall}(W, \hat{W}) = \frac{\big\|\{\hat{w} | \hat{w} \in \hat{W} \wedge \exists w \in W, \hat{w} \simeq w\}\big\|}{\|W\|} \tag{11}$$

In Equations 10 and 11, $\simeq$ denotes whether two tokens have similar semantics. SymLM [32] built a semantics word cluster trained on CodeSearchNet [29] and derived edit-distance based rules to measure the semantics similarity between tokens. We reuse their word cluster and rules.

GPT4Evaluator. Token-based metrics may not accurately reflect whether a name matches the program context or developers' intention. For example, the names `wait_sec` and `timeout` have no token overlap but denote similar semantics. On the other hand, existing work [57] on decompiled code summarization demonstrates that using GPT4 as an evaluator aligns better with human judgements than automatic metrics. Therefore, we adapt their method, further using GPT4 as an evaluator to measure the quality of generated names.

Specifically, we follow [57] and measure the quality of a name from *context relevance* and *semantics accuracy*. We query GPT4 per binary function. Each query consists of a decompiled function with the ground-truth variable names, and a name map from ground-truth names to predicted names. We ask GPT4 to first summarize the decompiled function, and evaluate each predicted name by answering two questions in scores from 1 (worst) to 5 (best): (1) Whether the predicted name is consistent with the program context? (2) Whether the predicted name accurately depicts the semantics of the variable? The prompt used and examples for each score are shown in Fig. 17 and Fig. 18 in the Appendix.

VI. Evaluation

A. Performance in terms of Semantics Match

We show the performance of both GenNm and VarBERT in Table I. We can see that overall, GenNm outperforms the baseline model VarBERT on both datasets. On the DIRTY dataset, GenNm outperforms VarBERT by 8.6 percentage points in terms of both precision and recall. On the VarCorpus dataset, GenNm outperforms VarBERT by 11.4 and 11.0 percentage points in terms of precision and recall, respectively. Note that the performance for VarBERT reproduced in Table I is lower than the reported statistics in the VarBERT paper. That is expected because we preclude potential overlap between training and test with a more strict setup. Appendix B shows that both GenNm and VarBERT achieve significantly higher performance on the subset of samples that have a high similarity to the training dataset (e.g., VarBERT and GenNm achieve a precision of 50.8% and 72.3% on the DIRTY dataset, respectively).

Moreover, we observe that complex projects typically contain more than one binaries. Different binaries in a project likely share similar coding styles or naming preferences. Therefore, a model may be able to predict better names if the corresponding project of a test program has been seen in the training dataset. Therefore, we further categorize the test programs by whether the corresponding projects are seen during training or not, noted as *project-in-train* and *project-not-in-train*. Note that this categorization is ***different*** from the *in-train* and *not-in-train* setup in DIRTY [16]. As pointed out by previous work [46], there are better solutions for renaming variables in functions that overlap with the training dataset (i.e., the “in-train” samples in DIRTY’s setup). On the other hand, in our setup, *project-in-train* mimics a realistic scenario that the naming style of an author group (e.g., an APT group [23]) is already learned beforehand, and a technique is used to analyze programs from the same author group. Both project-in-train and project-not-in-train samples ***do not overlap with the training data samples.***

We can see that both GenNm and VarBERT perform better on samples whose projects are seen during training. On those samples, GenNm outperforms VarBERT by more than 10 percentage points (on both datasets) in terms of both precision and recall. For the more challenging project-not-in-train samples, GenNm consistently outperforms the baseline model by 7.6–8.6 perecentage points, demonstrating better generalizability.

We train two versions of GenNm from code language models with different sizes, i.e., CodeGemma-2B [59] and CodeLlama-7B [51], denoted as GenNm$_{\text{CG-2B}}$ and GenNm$_{\text{CLM-7B}}$, using the DIRTY dataset. We can see that both models achieve better performance than the baseline model, demonstrating that our technique can generalize to models with different sizes. For other sections in the evaluation, GenNm refers to GenNm $_{\text{CG-2B}}$.

B. Performance Evaluated by GPT4Evaluator

We further use GPT4Evaluator to evaluate the performance of both models. Due to the limited budget, we randomly sample 500 functions (corresponding to 1632 variable names) from the DIRTY dataset. The results are shown in Fig. 11. We can see that in terms of both *context relevance* and *semantics accuracy*, GenNm achieves better scores than VarBERT. Especially, observe that for more than 50% variables, the names generated by GenNm are given scores of 4 or better for both measurements, indicating that GenNm can effectively recover high-level semantics information from decompiled code. It is also worth-noting that GenNm performs better in terms of context relevance than semantics accuracy. It indicates that GenNm can predict names within the correct program context in most time, yet it is more challenging to generate names that accurately reflect the semantics of ground-truth names. That is because compared to predicting names that are consistent with the program context, predicting the precise semantics of a variable entails a more accurate understanding for the semantics of the program, which is a challenging problem when the program does not have meaningful symbols [58]. We leave as future work to further improve the model’s understanding for decompiled code.

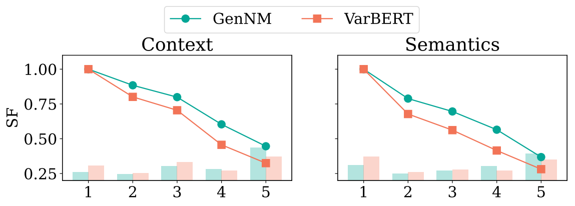


Fig. 11: Performance evaluated by the GPT4Evaluator. The two sub-figures show the scores for context relevance (Context) and semantics accuracy (Semantics), respectively. SF denotes the survival function. It indicates the number of samples achieving at least the corresponding score. The transparent bars reflect the distribution for each score.

C. Performance Compared to Blackbox LLMs

We compare the performance of GenNm with blackbox LLMs. We randomly sample 1000 functions from the DIRTY dataset and query two state-of-the-art blackbox LLMs, i.e., GPT-3.5 and GPT-4, with both the zero-shot and 3-shot setups. The prompts used are shown in Appendix C. The results are shown in Table II. Observe that GPT-4 achieves better performance than GPT-3.5, and both LLMs achieves better performance in the 3-shot setup. However, due to the distribution gap between decompiled functions and the pre-training knowledge of LLMs, both models underperforms GenNm. GenNm outperforms the best results achieved by blackbox LLMs by 11.3 and 6.4 percentage points in terms of precision and recall, respectively. Note that both LLMs are likely to be significantly (i.e., 10x–100x) larger in size than the base model used in GenNm. It demonstrates the necessity and effectiveness of fine-tuning a pre-trained code language model.

D. Generalizability

To measure the generalizability of GenNm, we analyze the performance of GenNm on variables with different ground-truth name frequencies and decompiled programs obtained with different setups.

Name frequency. We group variables by the frequency of their ground-truth names in the training dataset. The results are shown in Fig. 12. Observe that both models achieve better performance on names that appear more frequently in the training

TABLE I: Performance of GENNM compared with VarBERT. GENNM $_{\text{CG-2B}}$ and GENNM $_{\text{CLM-7B}}$ denote GENNM trained from CodeGemma-2B and CodeLlama-7B, respectively. Proj. NIT (Project Not-In-Train) denotes test programs whose corresponding *projects* are not seen in the training dataset. Proj. IT (Project In-Train) denotes test programs whose corresponding projects are seen in the training dataset. ***Both Proj. NIT and Proj. IT samples do not overlap with training data samples.***

Dataset	Model	Proj. NIT		Proj. IT		Overall	
		Precision	Recall	Precision	Recall	Precision	Recall
DIRTY	VarBERT	23.6	21.7	31.4	29.6	27.2	25.5
	GENNM $_{\text{CG-2B}}$	30.5	28.8	**41.7**	**39.6**	**35.8**	33.9
	GENNM $_{\text{CLM-7B}}$	**31.2**	**30.0**	39.7	38.6	35.2	**34.1**
VarCorpus	VarBERT	20.9	19.3	32.5	31.0	29.8	28.3
	GENNM $_{\text{CG-2B}}$	**29.5**	**27.4**	**44.7**	**42.8**	**41.2**	**39.3**

TABLE II: Performance compared to blackbox LLMs.

Model	Prompt	Precision	Recall
GPT-3.5	zero-shot	26.2	27.7
	3-shot	29.7	28.9
GPT-4	zero-shot	30.3	33.3
	3-shot	31.4	32.6
GENNM	-	**42.7**	**39.7**

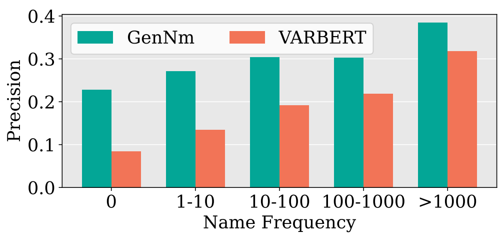


Fig. 12: Performance by name frequency on VarCorpus. The x-axis denotes the frequency of the ground-truth name for a variable in the training dataset of VarCorpus, and the y-axis the average precision achieved on the corresponding variables.

dataset. Note that GENNM consistently outperforms VarBERT on names with all name frequencies. Moreover, GENNM is more robust when the frequencies of names decrease. Especially, for names that are never seen in the training dataset, GENNM achieves a precision of over 20%, which is almost 2 times the performance of VarBERT on those variables. That indicates the generative model indeed generalizes better than a classification model on unseen names. Compared to the performance on frequent variables (i.e., variables with a name frequency higher than 1000), the performance of GENNM on rare variables (i.e., variables with a name frequency from 1 to 10) decreases by 29.4% (from 38.4% to 27.1%), while the performance of VarBERT decreases by 57.5% (from 31.8% to 13.5%). That indicates GENNM mitigates the biases of frequent names in the training dataset.

Setups to obtain decompiled functions. In other sections of the evaluation, we use decompiled functions emitted by IDA-Pro [30] from binaries compiled with the `-O0` flag. To measure the generalizability of GENNM, we further evaluate its performance on decompiled functions obtained in other setups. We run experiments on binary programs compiled with the `-O3` optimization flag since `-O3` is the most aggressive optimization option commonly used. Due to the limitation of computation resources, we do not evaluate on programs compiled with `-O1` and `-O2`. Additionally, we run experiments on the decompiled functions generated by Ghidra [22]. For both setups, we reuse the VarCorpus dataset following the same preprocess procedure discussed in Section V.

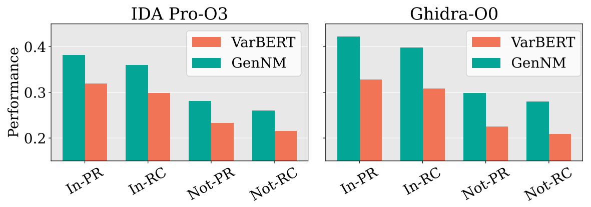


Fig. 13: Generalizability to other optimization levels and decompilers. *In-PR*, *In-RC*, *Not-PR*, and *Not-RC* denote the average *precision* and *recall* on samples whose project is seen or not seen in the train data, respectively.

The results are shown in Fig. 13. We can see that GENNM outperforms VarBERT in both setups. It demonstrates that GENNM can generalize to different setups. The improvement of GENNM on programs compiled with the `-O0` option is more significant than the improvement on programs compiled with `-O3`. That is because programs compiled with `-O3` are significantly longer and diverge further from the distribution of source due to the presence of aggressive optimizations. Therefore, it is more challenging for models to understand them, affecting the model's performance. We leave it as future work to further improve the model's capability of understanding programs compiled with aggressive optimization flags.

E. Ablation Study

We conduct ablation studies to analyze how each component contributes to the effectiveness of GENNM. Also, we study the effectiveness of different design decisions in constructing the symbol preference dataset. The results show that all components, i.e., the context-aware fine-tuning, SymPO, and program analysis based context augmentation, contribute to the effectiveness of GENNM. Moreover, we find that using ground-truth names to construct the symbol preference dataset results a model having worse performance than our design that constructs the preference dataset from model's predicted names. Also, our static feature based data selection heuristics reduce the size of dataset by 60%, while the model trained on the reduced dataset achieves slightly better results than the one trained on the whole dataset. Please see details in Appendix D.

VII. CASE STUDY

Examples of GENNM's prediction. To intuitively demonstrate the effectiveness of GENNM, we show examples of GENNM's prediction that receive each score in the GPT4Evaluator in Fig. 18 in the Appendix.

Malware reverse engineering. We use a real-world malware sample [38] to illustrate how GENNM helps a security analyst

```
1  v48 = recv(fd, v76, 0x1000, 0);
2  v76[v48] = 0;
3  for (i = strtok(v76, "\n");
     i && *i; i = strtok(0, "\n")) {
    //...`v52`:the last non-empty char in `i`
4   if (*i == ':') {
5    v53 = i;
     // parse the first part of the command
6    while (1) {
7     v54 = v53 - i;
8     if (v52 <= v53 - i) break;
9     v55 = *v53; v56 = v53++;
10    if (v55 == 32) goto LABEL_98;
11   }
12   v56 = v53;
13 LABEL_98:
14   *v56 = 0; v57 = v54;
     // stores the first part to `dest`
15   strcpy(&dest, i + 1); strcpy(i, &i[v57 + 1]);
16  }
   // find and execute the related command
17 for (j = (const char **)&unk_60A500;
        *j; j = v66 + 2) {
18  v73 = j;
19  v65 = strcasecmp(*j, &s2);
20  v66 = v73;
21  if (!v65) {
22   ((void(*)(int64, char *, char *))v73[1])
23                              (fd, &dest, i);
24 }}}
```

(a) Decompiled code output by IDA

```
1  nbytes = recv(fd, buf, 0x1000, 0);
2  buf[nbytes] = 0;
3  for (tok = strtok(buf, "\n");
     tok && *tok; tok = strtok(0, "\n")) {
    //...`lastidx`:the last non-empty char in `tok`
4   if (*tok == ':') {
5    p1 = tok;
     // parse the first part of the command
6    while (1) {
7     i_len_0 = p1 - tok;
8     if (lastidx <= p1 - tok)break;
9     c = *p1; lastchar_0 = p1++;
10    if (c == 32) goto LABEL_98;
11  }
12  lastchar_0 = p1;
13 LABEL_98:
14  *lastchar_0 = 0; i_len_1 = i_len_0;
    // stores the first part to `dest`
15  strcpy(&dest,tok+1);strcpy(tok,&tok[i_len_1 + 1]);
16 }
   // find and execute the related command
17 for (cmd_ptr = (const char **)&unk_60A500;
         *cmd_ptr; cmd_ptr = cmd_tmp + 2) {
18  matched_cmd = cmd_ptr;
19  cmp = strcasecmp(*cmd_ptr, &cmd);
20  cmd_tmp = matched_cmd;
21  if (!cmp) {
22    ((void(*)(int64, char *, char *))
23         matched_cmd[1])(fd, &dest, tok);
24 }}}
```

(b) Renamed code (generated names highlighted in orange)

Fig. 14: How GenNm helps security analyst understand a malware sample

reverse engineer a malware sample. Fig. 14 shows a code snippets from the studied real-world malware sample. It connects to a command-and-control (C&C) server, parses the command and dispatches the commands from the server. In Fig. 14a we show the decompiled code generated by IDA [30]. In Fig. 14b we show the corresponding code with variables renamed by GenNm. At lines 1–2, the malware receives commands from the server. Lines 3–15 parse the commands, and lines 17–24 dispatch and execute the commands.

We can see that the names predicted by GenNm makes the code snippets easier to understand. For example, `i` defined at line 3 is renamed to `tok`. It indicates that the variable stores a *token* of the command. At line 14, the variable `v57` is renamed to `i_len_1`. It indicates that the variable stores the length of a sub-component of the variable `i` (now renamed to `tok`). Therefore, it is easier to understand that lines 4–14 split a command to two parts and store them in `dest` and `tok`, respectively (line 15). More importantly, GenNm renames `j` at line 17 and `v73` at line 18 to `cmd_ptr` and `matched_cmd`, respectively. It reflects that lines 17–24 are dispatching and executing commands from the server. This would reveal the suspicious intention of this code snippet.

Binary summarization. We further study how GenNm helps the binary summarization task. We use GenNm to recover names in a decompiled function. Then we feed the function to ChatGPT and ask ChatGPT to summarize the decompiled function. The study shows that with the predicted variable names, ChatGPT captures more accurate information from the decompiled code. Details are in Appendix E.

VIII. Related Work

Recovering symbol names in decompiled code. There are existing efforts on reconstructing variable names in stripped binary programs [16], [34], [46], [6] leveraging the abstract syntax tree (AST) of programs [34], type information [16] recovered from decompiled code, and pre-training knowledge from source code [46]. State-of-the-art techniques [16], [46] formulate the problem as a classification task, whereas GenNm formulates it as a generative task. It can predicts names that rarely appear in the training dataset. There are recent work [63] leveraging ChatGPT [42] in this task. However, ChatGPT is majorly trained on source code and natural language corpus [11], [45], [44], and may not understand decompiled code effectively [31]. Besides recovering variable names, another stream of work focuses on recovering function names in decompiled code [32], [33]. Their efforts are complementary with us.

Reverse engineering. Existing efforts reverse engineer binary programs to analyze malware [64], [56], [5], harden programs [19] and facilitate fuzzing [20], [17], [52]. Their efforts are complementary with us, and the results of GenNm can benefit the reverse engineering tasks, as shown in Section VII.

Foundational binary program analysis. GenNm leverages existing foundational techniques [3], [9], [37], [12] to analyze binary programs, such as disassembly [41], type recovery [54], [35], [55], and decompilation [66]. State-of-the-arts achieve good performance in most cases [48], [8], [67].

IX. Conclusion

We propose a novel technique that leverages the strengths of generative models to recover meaningful variable names from the decompiled code of fully stripped binary programs. We employ symbol preference optimization to mitigate models' biases and use program analysis to alleviate hallucinations. Our prototype GenNm demonstrates significant improvements on SOTA in challenging setups.

References

[1] A. Al-Kaswan, T. Ahmed, M. Izadi, A. A. Sawant, P. Devanbu, and A. 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)*. IEEE, 2023, pp. 260–271.

[2] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida *et al.*, "Binrec: dynamic binary lifting and recompilation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[3] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 84–96.

[4] I. Angelakopoulos, G. Stringhini, and M. Egele, "FirmSolo: Enabling dynamic analysis of binary linux-based IoT kernel modules," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5021–5038. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/angelakopoulos

[5] S. Aonzo, Y. Han, A. Mantovani, and D. Balzarotti, "Humans vs. machines in malware classification," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[6] P. Banerjee, K. K. Pal, F. Wang, and C. Baral, "Variable name recovery in decompiled binary code using constrained masked language modeling," *arXiv preprint arXiv:2103.12801*, 2021.

[7] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, Q. V. Do, Y. Xu, and P. Fung, "A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity," 2023. [Online]. Available: https://arxiv.org/abs/2302.04023

[8] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation."

[9] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[10] A. Borji, "A categorical archive of chatgpt failures," 2023. [Online]. Available: https://arxiv.org/abs/2302.03494

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[12] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 353–368.

[13] K. Burk, F. Pagani, C. Kruegel, and G. Vigna, "Decomperson: How humans decompile and what we can learn from it," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2765–2782.

[14] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 555–565.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow bending: On the effectiveness of Control-Flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[16] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.

[17] J. Choi, K. Kim, D. Lee, and S. K. Cha, "Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 677–693.

[18] (2024) Cve-2018-4407. https://github.com/github/securitylab/tree/main/SecurityExploits/apple/darwin-xnu/icmp_error_CVE-2018-4407.

[19] G. J. Duck, Y. Zhang, and R. H. C. Yap, "Hardening binaries against more memory errors," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 117–131. [Online]. Available: https://doi.org/10.1145/3492321.3519580

[20] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 1–13.

[21] (2024) Gentoo packages. https://packages.gentoo.org/.

[22] (2024) Ghidra. https://ghidra-sre.org/.

[23] (2024) Groups mitre att&ck. https://attack.mitre.org/groups/.

[24] E. Gustafson, P. Grosen, N. Redini, S. Jha, A. Continella, R. Wang, K. Fu, S. Rampazzi, C. Kruegel, and G. Vigna, "Shimware: Toward practical security retrofitting for monolithic firmware images," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 32–45.

[25] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on knowledge and data engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[26] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *International Conference on Learning Representations*, 2019.

[27] C.-P. Hsieh, S. Sun, S. Kriman, S. Acharya, D. Rekesh, F. Jia, and B. Ginsburg, "Ruler: What's the real context size of your long-context language models?" *arXiv preprint arXiv:2404.06654*, 2024.

[28] C. Huang, Y. Li, C. C. Loy, and X. Tang, "Learning deep representation for imbalanced classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 5375–5384.

[29] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[30] (2023) A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro/.

[31] X. Jin, J. Larson, W. Yang, and Z. Lin, "Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models," *arXiv preprint arXiv:2312.09601*, 2023.

[32] X. Jin, K. Pei, J. Y. Won, and Z. 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*, 2022, pp. 1631–1645.

[33] H. Kim, J. Bak, K. Cho, and H. Koo, "A transformer-based function symbol name inference model from an assembly language for binary reversing," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 951–965.

[34] 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)*. IEEE, 2019, pp. 628–639.

[35] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[36] Y. Li, W. Xu, Y. Tang, X. Mi, and B. Wang, "Semhunt: Identifying vulnerability type with double validation in binary code." in *SEKE*, 2017, pp. 491–494.

[37] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, ser. CERIAS '10. West Lafayette, IN: CERIAS - Purdue University, 2010.

[38] (2024) Virustotal. https://www.virustotal.com/gui/file/03cfe768a8b4ffbe0bb0fdef986389dc.

[39] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "Re-mind:

a first look inside the mind of a reverse engineer," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2727–2745.

[40] J.-P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro, "Dynamically checking ownership policies in concurrent c/c++ programs," *ACM Sigplan Notices*, vol. 45, no. 1, pp. 457–470, 2010.

[41] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1187–1198.

[42] OpenAI, "Chatgpt," 2023. [Online]. Available: https://chat.openai.com/chat

[43] ——, "Gpt-4 technical report," 2023.

[44] ——, "Introducing chatgpt," 2023. [Online]. Available: https://openai.com/blog/chatgpt

[45] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 27 730–27 744. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf

[46] K. K. Pal, A. P. Bajaj, P. Banerjee, A. Dutcher, M. Nakamura, Z. L. Basque, H. Gupta, S. A. Sawant, U. Anantheswaran, Y. 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)*. IEEE Computer Society, 2024, pp. 152–152.

[47] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. 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*, 2021, pp. 690–702.

[48] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "Xda: Accurate, robust disassembly with transfer learning."

[49] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[50] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[51] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[52] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise mmio modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1239–1256.

[53] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using logic programming to recover c++ classes and methods from compiled executables," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 426–441.

[54] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[55] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[56] C. Spensky, H. Hu, and K. Leach, "LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis," February 2016.

[57] Z. Su, X. Xu, Z. Huang, K. Zhang, and X. Zhang, "Source code foundation models are transferable binary analysis knowledge bases," *arXiv preprint arXiv:2405.19581*, 2024.

[58] Z. Su, X. Xu, Z. Huang, Z. Zhang, Y. Ye, J. Huang, and X. Zhang, "Codeart: Better code models by attention regularization when symbols are lacking," *arXiv preprint arXiv:2402.11842*, 2024.

[59] C. Team, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[60] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[61] Y. Wang, X. Xu, P. Wilke, and Z. Shao, "Compcertelf: verified separate compilation of c programs into elf object files," vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428265

[62] H. Wen and Z. Lin, "Egg hunt in tesla infotainment: A first look at reverse engineering of qt binaries," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3997–4014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wen

[63] X. Xu, Z. Zhang, S. Feng, Y. Ye, Z. Su, N. Jiang, S. Cheng, L. Tan, and X. Zhang, "Lmpa: Improving decompilation by synergy of large language model and program analysis," 2023.

[64] Z. Xu, A. Nappa, R. Baykov, G. Yang, J. Caballero, and G. Gu, "Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 179–190.

[65] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 462–472.

[66] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." in *NDSS*. Citeseer, 2015.

[67] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, "D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2391–2408.

[68] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 813–832.

APPENDIX

A. How a Reference Model Prevents the SymPO Model from Diverging too much

We show the gradient of the SymPO loss in Equations 12–14. As shown in Equation 12, the gradient is the multiplication of two terms. The second term in the bracket is not affected by the reference model, and is straightforward: it enlarges the probability for better names while decreases the probability for worse names.

On the other hand, the first term constraints the magnitude of the gradient for a given data sample. It can be equally transformed to Equation 14. Observe that if the model being optimized already shows significant preference towards the better names compared with the reference model, this term will be close to zero. The updates (to the model weights) introduced by the corresponding data sample will thus be smaller. Therefore, the reference model reduces further optimization on the already learned preference, minimizing the divergence from the reference model.

B. Dataset Preprocessing and Statistics

Preprocessing following the DIRTY dataset. We use GHCC to compile C/C++ projects on GitHub created in 2012-2022. Different from DIRTY, (1) we additionally filter out projects with less than 20 stars for the quality consideration. (2) We only include executable binary programs in our dataset, precluding intermediate relocatable binary files since the semantics of a relocatable file relies on its symbol table [61], which may be stripped away.

Rationale of deduplicating binaries by function names. In our preprocessing pipeline, we conservatively deduplicate binaries by including a binary program only if more than 70% of its function names are not in the dataset yet. It is common that a project puts the main logic in the shared object (.so) file and keep other binaries as simple wrapper programs. Take the tool Bibutils [2] as an example. The (corresponding source code files of) two binary programs `xml2ris` [3] and `xml2end` [4] are simply two wrapper programs for a shared object `libbibutils.so`. All three binary programs are in the original VarCorpus dataset. However, after including the shared object in the dataset, it is not beneficial to include the two wrapper programs. As a result, as shown in Table III, both our processed datasets are smaller than the original VarCorpus dataset, while their diversity are better than the original VarCorpus dataset.

Checking data leakage with string similarity. We propose to use string similarity, instead of exact string matching to identify data leakage in the test dataset (i.e., test functions that present in the training dataset). Previous work [46], [16], [34] consider two functions as the same only when their normalized string are exactly the same. For example, the VarBERT authors deduplicate the VarCorpus dataset so that all the functions in VarCorpus are not exactly the same. Therefore, there is no overlap between the training and test data samples in terms of exact string match. However, we observe that considering a sample as data leakage only when there is an exactly matched string in the training data still significantly overestimates the performance of a tested model.

For example, we observe that there are 15,363 functions named `allocate` in the *deduplicated* VarCorpus dataset. We show three of them in Fig. 16 to illustrate the problem. All three versions allocate a chuck of memory and terminates the execution on failure. The two versions in (a) and (b) are only different in the size of allocation. They are almost the same function, but cannot be captured by exact string match even after normalization. Suppose that version (a) is in the training dataset. The performance of a model on version (b) cannot really reflect the generalizability of the model. On the other hand, the third function has semantics differences in that it explicitly sets the allocated memory to zero. Therefore, simply considering all functions with the same name as potential data leakage may introduce significant false positives.

We propose to use string similarity [5] as the metric to conservatively check potential data leakage. The string similarity is calculated based on string edit distance, ranging from 0 (indicating two strings have no overlap) to 100 (indicating two strings are exact match). Empirically, we consider a test sample as overlapped with training dataset if the highest string similarity of the sample is larger than 90 to a training data sample. Table IV shows the performance of GENNM and VarBERT on data samples whose highest string similarity to a training data sample is larger than 90. We can see that the performance of all models are substantially better than the performance shown in Table I. The performance, unfortunately, cannot faithfully reflect the capability of models on the variable recovery problem.

TABLE III: Dataset statistics. Each column denotes a dataset. *#Func* denotes the total number functions in the dataset. *Unique Funcs* denotes the ratio of functions with unique function names. *Unique Name List* denotes the ratio of functions with unique name lists of variables. *#Vars* denotes the total number of variables, and *Unique Names* denotes the ratio of variables with unique variable names.

	VarCorpus-Ori	VarCorpus-Our	DIRTY-Our
#Func	**1,995,847**	895,004	348,213
Unique Funcs (by name) (%)	46.8	81.3	**89.4**
Unique Name List (%)	29.6	**52.7**	40.4
#Vars	**6,126,592**	3,363,688	1,156,214
Unique Names (%)	6.5	9.8	**12.2**

TABLE IV: Performance of models on functions whose highest string similarity score in the training dataset is larger than 90.

Dataset	Model	PR	RC
DIRTY	VarBERT	50.8	50.6
	GENNM $_{\text{Gemma2B}}$	59.7	58.6
	GENNM $_{\text{CLM7B}}$	**72.3**	**71.8**
VarCorpus	VarBERT	44.4	43.7
	GENNM $_{\text{Gemma2B}}$	**56.1**	**55.1**

C. Prompts input to ChatGPT

For each model, we start a query with a prompt describing the task and output format, as follows:

[2]https://github.com/biodranik/bibutils

[3]https://github.com/biodranik/bibutils/blob/master/bin/xml2ris.c

[4]https://github.com/biodranik/bibutils/blob/master/bin/xml2end.c

[5]https://anhaidgroup.github.io/py_stringmatching/v0.3.x/Ratio.html

$$\nabla_\Theta \mathcal{L}_{\text{SymPO}}(\Theta,\Theta_{\text{ctx}}) ::= -\beta \mathbb{E}_{(q,b,w)\sim D_{\text{prf}}}\Bigg[\delta(q,b,w,\Theta,\Theta_{\text{ctx}})\Big[\underbrace{\nabla_\Theta \log \mathbf{P}(b|q;\Theta)}_{\substack{\text{increase preference}\\ \text{towards \textbf{\textit{better}} symbols}}} - \underbrace{\nabla_\Theta \log \mathbf{P}(w|q;\Theta)}_{\substack{\text{decrease preference}\\ \text{towards \textbf{\textit{worse}} symbols}}}\Big]\Bigg] \tag{12}$$

$$\delta(q,b,w,\Theta,\Theta_{\text{ctx}}) ::= \sigma\Big(\beta \log \frac{\mathbf{P}(w|q;\Theta)}{\mathbf{P}(w|q;\Theta_{\text{ctx}})} - \beta \log \frac{\mathbf{P}(b|q;\Theta)}{\mathbf{P}(b|q;\Theta_{\text{ctx}})}\Big) \tag{13}$$

$$= \sigma\Bigg(\beta\Big(\log \frac{\mathbf{P}(w|q;\Theta)}{\mathbf{P}(b|q;\Theta)} - \log \frac{\mathbf{P}(w|q;\Theta_{\text{ctx}})}{\mathbf{P}(b|q;\Theta_{\text{ctx}})}\Big)\Bigg) \tag{14}$$

$$\frac{\begin{array}{c} f_1 \in B.\textbf{\textit{funcs}} \qquad id_0 = args[i] \qquad p_0 = r[i] \qquad r,_{=} f_1.\textbf{\textit{body}} \\ n_1 = N_{in}[f_1.\textbf{\textit{fid}}][p_0] \qquad \sigma' = \sigma\big[f.\textbf{\textit{fid}}, id_0 \rightsquigarrow \{(f_1.\textbf{\textit{fid}}, p_0, n_1)\} \cup \sigma[f.\textbf{\textit{fid}}][id_0]\big] \end{array}}{\langle\cdot,\sigma\rangle \vdash f_1.\textbf{\textit{fid}}(args) : \langle\cdot,\sigma'\rangle}\textbf{Call-R}$$

$$\frac{\begin{array}{c} f_1 \in B.\textbf{\textit{funcs}} \qquad id_1 := f.\textbf{\textit{fid}}(...) \in f_1.\textbf{\textit{body}} \\ n_1 = N_{in}[f_1.\textbf{\textit{fid}}][id_1] \qquad \sigma' = \sigma\big[f.\textbf{\textit{fid}}, id_0 \rightsquigarrow \sigma[f.\textbf{\textit{fid}}][id_0] \cup \{(f_1.\textbf{\textit{fid}}, id_1, n_1)\}\big] \end{array}}{\langle\cdot,\sigma\rangle \vdash \textbf{return } id_0 : \langle\cdot,\sigma'\rangle}\textbf{Ret-R}$$

$$\frac{\langle\cdot,\sigma\rangle \vdash \mathcal{S}_1 : \langle\cdot,\sigma_1\rangle \qquad \langle\cdot,\sigma_1\rangle \vdash \mathcal{S}_2 : \langle\cdot,\sigma_2\rangle}{\langle\cdot,\sigma\rangle \vdash \mathcal{S}_1;\mathcal{S}_2 : \langle\cdot,\sigma_2\rangle}\textbf{Step} \qquad \frac{\langle\cdot,\sigma\rangle \vdash S : \langle\cdot,\sigma\rangle}{\langle\cdot,\sigma\rangle \vdash \textbf{while}(\mathcal{E})\{\mathcal{S}\} : \langle\cdot,\sigma\rangle}\textbf{While}$$

$$\frac{\begin{array}{c} \langle\cdot,\sigma\rangle \vdash \mathcal{S}_1 : \langle\cdot,\sigma_1\rangle \qquad \langle\cdot,\sigma\rangle \vdash \mathcal{S}_2 : \langle\cdot,\sigma_2\rangle \qquad \big((fid,vid) \in \sigma_1 \vee (fid,vid) \in \sigma_2\big) \\ \sigma_3[fid][vid] = \sigma_1[fid][vid] \cup \sigma_2[fid][vid] \end{array}}{\langle\cdot,\sigma\rangle \vdash \textbf{if}(\mathcal{E})\{\mathcal{S}_1\}\textbf{else}\{\mathcal{S}_2\} : \langle\cdot,\sigma_3\rangle}\textbf{If} \qquad \frac{}{\langle\cdot,\sigma\rangle \vdash \textit{Other Stmts} : \langle\cdot,\sigma\rangle}\textbf{Default}$$

Fig. 15: Other Correlation Extraction Rules

```
void *__fastcall allocate(unsigned int n)
{
  void *v1;

  v1 = 0LL;
  if ( n )
  {
    v1 = calloc(1uLL, n);
    if ( !v1 )
      no_space();
  }
  return v1;
}
```

(a) A version of `allocate`

```
void *__fastcall allocate(int n)
{
  void *v1;

  v1 = 0LL;
  if ( n )
  {
    v1 = calloc(1uLL, (n + 10));
    if ( !v1 )
      no_space();
  }
  return v1;
}
```

(b) A version almost the same with (a)

```
void *__fastcall allocate(size_t n)
{
  void *p;

  p = malloc(n);
  if ( !n )
    error_exit("Memory allocation failure");
  memset(n, 0, n);
  return n;
}
```

(c) A version different from (a)

Fig. 16: Three versions of `allocate`. They demonstrates why checking data leakage with exact string match may still overestimate models' performance. Versions (a) and (b) are almost the same. The difference is highlighted. Version (c) is different from both (a) and (b) because it has different semantics, e.g., setting the allocated memory to zeros. On the other hand, string-similarity can capture the similarity between (a) and (b) while distinguish them with (c). The string similarity scores between (a) and (b), (a) and (c), (b) and (c) are 95, 58, and 58, respectively.

Prompt: You are a helpful binary program expert. You are helping the user to understand the binary program below. You will suggest meaningful names for the variables and functions the user asks about. The asked identifiers are specified in the format of `Q:[var1,var2,...]` You will suggest one name for each asked identifier. You must output the suggested names in the json format: `{"var1": "suggested_name1", "var2": "suggested_name2", ...}`

We evaluate each model with both 0-shot and 3-shot settings. In a 0-shot experiment, we simply follow the prompt with a decompiled function to query. In a 3-shot experiment, we gives each model 3 "examples" before each query. In each query, we randomly sample 3 decompiled functions from the training dataset. Then we input both the sampled decompiled functions and the expected output of these functions. After that, we send to the model the query function.

D. Details for Ablation Study

There are three major components in GENNM, the context aware fine-tuning, the symbol preference optimiza-

System prompt:
You are an experienced C/C++ reverse engineer. Please act as an impartial judge and evaluate the quality of names of variables in the given decompiled program. You will be provided with (1) the decompiled code with ground-truth variable names, (2) the variable names predicted by an AI assistant.
In the evaluation, you should answer the following questions:
A. Does the variable name reflect relevant context (domain)? Answer the question in range 5(best) to 1(worst).
Domain/context describes the high-level program context of the variable. It is more of the general high-level domain (e.g., network, memory, CPS, physics, GUI, etc) rather than specific semantics (e.g., len, size, pointer).

- For 5, the predicted name and the ground-truth name should describe the same domain/context. Or, both the predicted name and the ground-truth name does not explicitly mention a specific domain.
- For 4, the domains of the predicted name and the ground-truth name should be similar and relevant, although may not be exactly the same. The predicted name domain may be a superset or subset of the ground truth. The predicted domain may be closely related to the ground-truth domain. The predicted name and ground-truth name may be two different perspectives of a same specific domain.
- For 3, the predicted name does not explicitly mention a specific context, but the ground-truth name does. The predicted name only contains low level operations. From the predicted name, one cannot deduce the high-level purpose of the decompiled function/variable.
- For 2, the predicted name is slightly misleading. The domain for predicted name is different and not relevant to the ground-truth domain. However, although misleading, the domain is only implied by the choice of words, and is not explicitly mentioned.
- For 1, the predicted name is completely misleading. The name is irrelevant to the ground-truth domain, and it is explicitly mentioned in the name.

B. Does the predicted name reflect relevant semantics? Answer the question in range 1(best) to 5(worst).
Semantics means the specific high-level meanings denoted by a variable (e.g., len, errmsg, file).

- For 5, the semantics of the name should be almost exactly the same to the ground truth. Or, both the predicted name and the ground-truth name do not have meaningful semantics.
- For 4, the semantics of the predicted name are similar to the ground-truth name. It may be vague, but the overall semantics and purpose is correct.
- For 3, the predicted name does not specify meaningful semantics but the ground truth name does. It only indicates some low-level operations without high level abstractions.
- For 2, the summary specify relevant but inaccurate semantics. The semantics specified in the predicted name may be relevant to the ground truth, but they have significant differences.
- For 1, the summary contains irrelevant semantics. It denotes a totally different semantics with the ground-truth.

You should first briefly summarize the provided decompiled code, then for each predicted variable name, follow the workflow:
Step1: Output the placeholder variable name you are analyzing, and its ground truth name.
Step2: Explain the ground truth name. (Why it is named like that? What is the high-level context? What is the high-level semantics?)
Step3: Output the predicted name, and explain it.
Step4: Output your score in the format:
```
{'var': (ground-truth name here), 'prediction': (predicted name here), 'score': {'Q-A':
[1-5], 'Q-B': [1-5]}}
```
Repeat the process for each variable name in the predicted name map.
User prompt:
Decompiled code with ground-truth variable names: ...
Predicted variable names: ...

Fig. 17: Prompts to GPT4Evaluator

TABLE V: Hyper-parameters in GenNm

Model	Parameter	Value
Gemma-2B	batch size	128
	learning rate scheduler	cosine
	learning rate	5e-5
	warmup steps	2000
CodeLlama-7B	batch size	64
	learning rate scheduler	cosine
	learning rate	5e-5
	warmup steps	2000
SymPO(Both)	batch size	64
	learning rate scheduler	cosine
	learning rate	1e-6
	warmup steps	100
Generation(Both)	temperature	0.2
	top_k	50

TABLE VI: How each component in GenNm contributes to the results. Ctx and SymPO denote GenNm with only context-aware fine-tuning and GenNm further tuned with SymPO. Analysis and No-Analysis denote whether to augment contexts via program analysis or not.

	Precision		Recall	
	Ctx	SymPO	Ctx	SymPO
No-Analysis	34.0	35.8	33.4	35.2
Analysis	36.3	**36.8**	34.6	**35.3**
VarBERT	26.5		25.7	

tion (SymPO), and the program analysis based context augmentation. We study the contribution of each components in Table VI. Specifically, *Ctx* denotes the model that is trained with only the context-aware fine-tuning. *SymPO* denotes the model that is further trained with symbol preference optimization. Note that SymPO starts from a context-aware fine-tuned model. Therefore it is not feasible to train a model with only SymPO. Also note that the program analysis based context augmentation can be added to both models. We can see that both *Ctx* and *SymPO* outperforms VarBERT, and *SymPO* further improves over *Ctx*, demonstrating the effectiveness of both components. On the other hand, observe that program analysis based context augmentation improves both models' performance.

We further study how each design decision in constructing the symbol preference dataset affects the performance. The results are shown in Table VII. The default setup (shown in the row *SymPO*) uses the best names predicted by the model as preferred names, and uses static feature based heuristics to reduce the size and noise of the constructed dataset. The second row (*SymPO w/o Data Filtering*) shows the dataset constructed without the static feature based heuristics. The third row (*SymPO w/ ground-truth Names*) shows the dataset that uses the ground-truth names as the preferred names. We can see that the static feature based heuristics reduce the dataset size by around 60%. And it demonstrates slightly better overall performance than training a model on the whole dataset. On the other hand, observe that the dataset constructed from ground-truth names results significantly worse performance than the default setup.

E. Case Study: Binary Summarization

Binary summarization helps a reverse engineer understand a binary function by generating natural language summary for the function. [1], [31]. Existing work show that the state-of-the-art code models still have gaps understanding decompiled code due to the lack of symbols [31]. We show an example in Fig. 19 to illustrate how names generated by GENNM benefit the binary summarization downstream task. We show the source code, the decompiled code, and the renamed code in Fig. 19a, 19b, and 19c, respectively. The code snippet initializes a BMP image structure. It calculates the absolute values of the width and height (lines 2–3 in Fig. 19a), calculates the row size of the image (line 4), allocates a chunk of memory for the image (line 5), and returns a value indicating whether the initialization succeeds (line 6).

We input the three versions to ChatGPT and show part of its response below. We use numbers to label the sentences and use red texts to emphasize extra information obtained from the renamed code (compared to the decompiled code). The full response from ChatGPT are shown as follows:

Source code: This function, `bmp_create`, ❶*initializes a BMP structure* with specified width `w` and height `h`. ❷It sets the *width and height* of the BMP structure to the *absolute values* of `w` and `h`, ❸ calculates the `stride` (which is the *aligned width considering a 4-byte boundary*), and ❹ *allocates memory* for the pixel data (`pdata`). The function ❺ *returns 0 if memory allocation is successful, and -1 otherwise*.

Decompiled code: This function, `sub_401580`, ❶ takes three arguments and performs a series of operations on them. ❷ It calculates the absolute values of a2 and a3 and stores them at specific memory locations relative to a1. ❸ It then calculates a value based on these absolute values and stores it at another location. ❹ Finally, the function attempts to allocate memory proportional to the calculated value and a3. ❺ If the memory allocation is successful, the function returns 0; otherwise, it returns -1. This suggests the function is used for setting up a data structure with dynamically allocated memory.

Renamed code: This function, `sub_401580`, ❶ initializes an image structure. It takes three parameters: a pointer to the image structure img, and two integers w and h for width and height. ❷ The width and height are stored as absolute values in the image structure. ❸ It then calculates and stores the row size, aligning it to a 4-byte boundary. ❹ Memory is allocated for the image data, the size of which is based on the calculated row size and height. ❺ The function returns 0 if the memory allocation is successful, or -1 (indicating an error) if the allocation fails.

We can see that the description for the renamed code is closer to that of the source code. It is more relevant to the context (e.g., in ❶ "initializes an image structure" vs "setting up a data structure") and is more accurate at the details (e.g., in ❸, the description for the decompiled version misses the "4-byte boundary alignment").

TABLE VII: Effectiveness of design decisions in SymPO

Dataset	#Pairs	Proj. not in train		Proj. in train		Overall	
		Precision	Recall	Precision	Recall	Precision	Recall
SymPO	94.3k	32.4	30.8	**42.3**	**40.6**	**36.2**	**34.6**
SymPO w/o Data Filtering	232k	**32.9**(+0.5)	**31.1**(+0.4)	40.3(-1.9)	38.6(-2.0)	35.8(-0.5)	34.0(-0.5)
SymPO w/ ground-truth Names	93.1k	31.0(-1.5)	29.5(-1.3)	40.5(-1.8)	39.0(-1.6)	34.7(-1.6)	33.2(-1.4)

```
__int64 process_wait(__pid_t *proc,
              unsigned int wait_secs){
 ...
 memset(&sigact, 0, sizeof(sigact));
 sigact.sa_handler = (__sighandler_t)sighandler;
 sigaction(14, &sigact, 0LL);
 alarm(wait_secs);
 if(waitpid(*proc, &status, 0) == *proc)
   ...
 return result;
}

Prediction: timeout
```

(a) Context:5, Semantics:5. The predicted name has exactly the same semantics and context with the ground-truth name.

```
__int64 randn(double mu, double sigma){

  if ( ... ){
   v2 = (double)qword_4138D0 * sigma + mu;
  }}...

Prediction: variance
```

(b) Context:4, Semantics:2. The predicted name has almost the same context as the ground truth (both are related to statistics). However, the semantics is misleading since variance is typically the square of sigma.

```
__int64 file_exists(const char *filename)
{
    ...
    fd = open(filename, 0);
    if (fd < 0)
        return (unsigned int)fd;
    close(fd);
    return 1LL;
}

Prediction: path
```

(c) Context:5, Semantics:4. The predicted name is consistent with the program context. However, the semantics of the predicted name does not imply the variable refers to a file. (`path` may also point to a directory.)

```
_QWORD *__fastcall udps_add(char* sess){
  ...
  n = malloc(0x40uLL);
  if (n){
    memcpy(n, &sess, 0x40uLL);
      ...
    }else{
        perror("malloc");
        ...
}

Prediction: data
```

(d) Context:3, Semantics:3. The predicted name does not imply any specific program context, while the ground-truth name `sess` has specific contexts about network. Similarly, the predicted name does not reflect the semantics of this variable, which denotes a "session".

```
__int64 ot_accept_client(int fd){

  ipstr = 0;
  if (...)
    ipstr = dns_query(...);
  if ( !ipstr )
    ...
}

Prediction: dom
```

(e) Context:2, Semantics:2. The predicted name `dom` is likely an abbreviation for 'domain'. Although it is in general related to network programming, the implied context is not accurate since the ground-truth name implies context about network address. Also, the predicted semantic is misleading since the variable denotes an IP address string, not a domain.

```
_QWORD *obfsem_randomize_decode(_BYTE *in,
  __int64 insiz, _QWORD *out, size_t *outsiz){

  qword_409670 = malloc(*outsiz);
  for (i = 0; i < *outsiz; ++i){
    *((_BYTE *)qword_409670 + i)
      = *(in + i % 4) ^ *(in + i + 6);

    *(qword_409670 + i)
     ^= *(off_4091C8 + i % 4);
  }
  result = out;
  *out = qword_409670;
  return result;

Prediction: key
```

(f) Context:1, Semantics:1. The predicted name is completely misleading and irrelevant to the program context. We speculate GENNM predicts the name `key` because it associates the bit-wise operations in the loop with encryption.

Fig. 18: Examples of GENNM's prediction for each score of GPT4Evaluator. ground-truth names are highlighted in green and predicted names are highlighted in orange.

```
int bmp_create(BMP *pb,int w,int h){
 pb->width = abs(w);
 pb->height = abs(h);
 pb->stride = ALIGN(pb->width*3,4);
 pb->pdata = malloc((size_t)pb->stride * h);
 return pb->pdata ? 0 : -1;
}
```

(a) Source code

```
int64 sub_401580(int64 a1,int a2,int a3){
 int64 result;
 *(int *)a1 = abs32(a2);
 *(int *)(a1 + 4) = abs32(a3);
 *(int *)(a1 + 8) = 4 *
      (3 * (*(int *)a1 + 1) / 4);
 *(int64 *)(a1 + 16) =
      malloc(*(int *)(a1 + 8) * (int64)a3);
 if (*(int64 *)(a1 + 16)) result = 0;
 else result = -1;
 return result;
}
```

(b) Decompiled code

```
int64 sub_401580(int64 img, int w, int h) {
 int64 result;
 *(int *)img = abs32(w);
 *(int *)(img + 4) = abs32(h);
 *(int *)(img + 8) = 4 *
      (3 * (*(int *)img + 1) / 4);
 *(int64 *)(img + 16) =
      malloc(*(int *)(img + 8) * (int64)h);
 if (*(int64 *)(img + 16)) result = 0;
 else result = -1;
 return result;
}
```

(c) Renamed code (generated names highlighted in orange)

Fig. 19: Binary summarization