Unleashing the Power of Generative Model in 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
Purdue University
{xzx, zhan3299, su284, huan1562, feng292, ye203, jiang719, xie342, cheng535, lintan}@purdue.edu
xyzhang@cs.purdue.edu

***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. We build a prototype, GenNm, from pre-trained generative models CodeGemma-2B, CodeLlama-7B, and CodeLlama-34B. We fine-tune GenNm on decompiled functions and teach models to leverage contextual information. GenNm includes names from callers and callees while querying a function, providing rich contextual information within the model's input token limitation. We mitigate model biases by aligning the output distribution of models with symbol preferences of developers. Our results show that GenNm improves the state-of-the-art name recovery precision by 5.6–11.4 percentage points on two commonly used datasets and improves the state-of-the-art by 32% (from 17.3% 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 [61], maintaining and hardening legacy software [13], [41], [12], detecting vulnerabilities in commercial-off-the-shelf software [68], [37], [64], [57], [63], and analyzing malware that threatens our daily lives [67], [55], [5]. A significant challenge, however, is presented by the fact that most of 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 [7], [47], [42], [2], [70], function boundary identification [3], type inference [38], [53], [36], [54], [71], [46], recovery of high-level abstractions [52], [32], [65], code structure [6], and data structures [52] 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 [10], [69] 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 [35], [15], [45]. 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], [29]. (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 [15] or Linux packages [45], 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.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240276
www.ndss-symposium.org

To address the challenges, we propose GENNM as a systematic solution to recovering names from fully stripped binaries. GENNM leverages a generative code language model fine-tuned with *contextual information* and *symbol preferences*. It composes variable names from tokens, and thus has better chance to 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. Our evaluation results in Section VI-D show that more than 95% rare names are composed from commonly appeared tokens.

To leverage the power of a generative model, a typical fine-tuning technique such as that in DIRE [35] and ReSym [62] simply guides a model to predict names from individual binary functions. Consequently, the trained model has limited knowledge about how to leverage contextual information. Nevertheless, information from the calling context plays a crucial role in understanding binary programs, as the absence of meaningful function names hinders the model's ability to deduce the semantics of calling contexts. We thus propose a novel context-aware fine-tuning paradigm that teaches a model to reason a binary function with both the function body and the information from its calling contexts. The model thus can learn the relation between names of local variables and names in the calling contexts.

Moreover, 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 ones under given program contexts.

We design the inference stage of GENNM as an iterative process. It is inspired by recent studies on human reverse engineering [40], [11], which emphasize the practice of inspecting all functions in a breadth-first manner, followed by iterative refinement. Initially, GENNM generates variable names for each function based on local context (i.e., the function body). Next, we traverse the program call graph to propagate contextual information from each function to its caller and callee functions, aligning with the training objective. Inspired by previous work [62], we leverage program analysis and majority voting to pick the final name across different iterations.

We summarize our contributions as follows:

- We propose a novel context-aware fine-tuning paradigm that teaches a model to leverage contextual information when reasoning a decompiled function.
- 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 design an iterative inference process aligned with the way human reverse engineers leverage contextual information.
- We develop a prototype GENNM (Unleashing the Power of Generative Model in Recovering Variable Names from Stripped Binary)[1]. GENNM improves the name recovery accuracy over the state-of-the-art techniques [45], [62] by 5.6–11.4 percentage points on two commonly used datasets with the most challenging setup. On challenging cases where the ground-truth names are not seen during training, GENNM improves over the state-of-the-art techniques [45], [62] by 168% (8.5% versus 22.8%) and 32% (17.3% versus 22.8%), respectively.

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 [17]. 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 [31] 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.

[1] Available at https://doi.org/10.5281/zenodo.14220042

```
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.

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

The state-of-the-art technique VarBERT [45] leverages the Transformer model [59] 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 [2]. 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 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], [29]. 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,

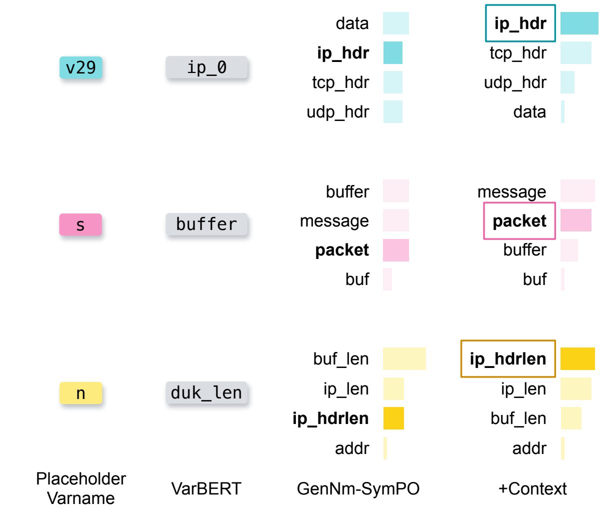


Fig. 2: Name selections for baseline (VarBERT) and name distributions for the predictions of GenNm. Each column denotes the predictions of a technique. VarBERT denotes the baseline model, *GenNm-SymPO* denotes the GenNm model after fine-tuning and symbol preference optimization. *+Context* denotes the model is used with the contextual information propagated along the call graph. 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.

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

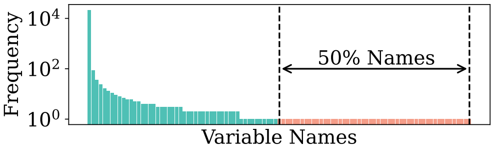


Fig. 3: Distribution of name frequencies. More than 50% variable names (in orange) appear only once in the training dataset.

the model biases towards the name `buffer` after seeing the variable is used as the first argument to `memset` in the query. Please see Appendix C for a quantified statistical test.

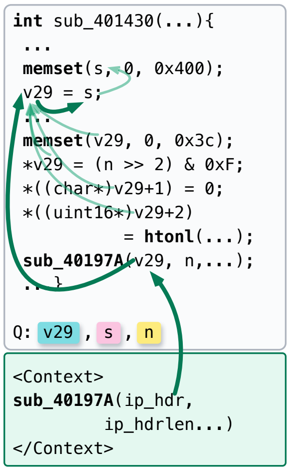
Challenge 3: Missing contextual information makes prediction difficult. Limited by the input length and the understanding capability of typical classification models (which are smaller than pre-trained generative models), VarBERT and many other existing works [35], [15] analyzes only one function at 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 augment a query function with better context, we propagate information of individual functions through the call graph. We design a new context-aware fine-tuning paradigm to teach the generative model how to predict names considering additional contextual information. To accommodate the generative model to the biased distribution of variable names, we design *symbol preference optimization* that aligns the model with the symbol preference of developers.

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. Large language models (LLMs) (e.g., ChatGPT and GPT-4 [44], [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. Our evaluation in Section VI-F shows that ChatGPT and GPT-4 underperform 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.



```
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 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 is 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 perform 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) [49] 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).

Solution for Challenge 3: Iterative inference and context-aware fine-tuning. 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](detailed discussion is in Appendix F of an extended version of this paper [66]). 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 (+Context) 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.

A generative model fine-tuned with only the function body and the ground-truth names (as shown in Fig. 1c) may have limited knowledge about how to effectively leverage the contextual information. We therefore design a novel context-aware fine-tuning paradigm, providing contextual information (as shown in the green box in Fig. 4) during fine-tuning so that the model can learn the relation between the names of local variables and the names in the calling contexts. According to our experiments in Section VI-G, this is the key reason for GenNm's superior performance.

Finally, to select the best name across multiple inference iterations, we propose a name validation algorithm to select (from top-ranked candidates) the name that is most consistent with the local program context. We 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.

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

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

Fig. 5: Formal definitions of the problem

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 in 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 the 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 in 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

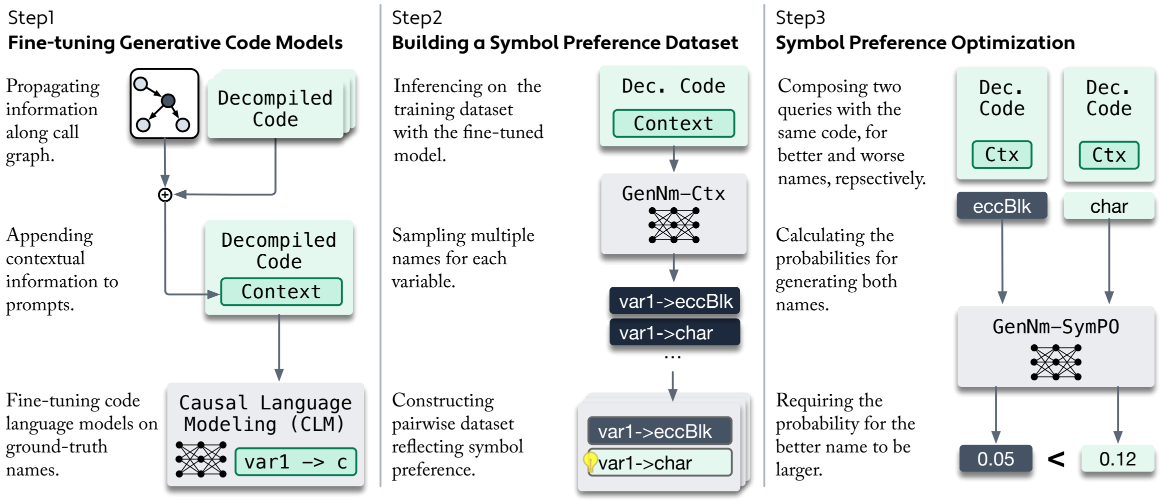

Step1
Fine-tuning Generative Code Models
Propagating information along call graph.
Decompiled Code
Appending contextual information to prompts.
Decompiled Code
Context
Fine-tuning code language models on ground-truth names.
Causal Language Modeling (CLM)
var1 -> c
Step2
Building a Symbol Preference Dataset
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

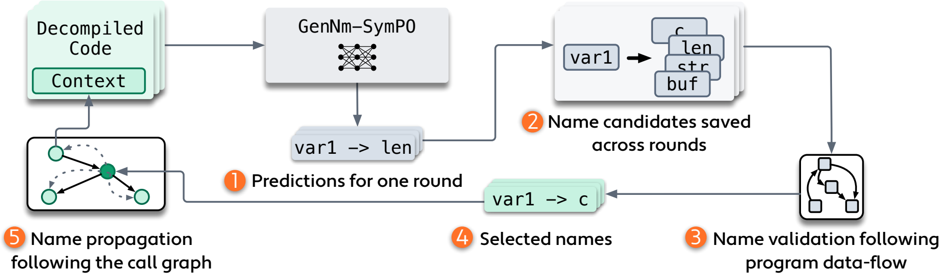

Decompiled Code
Context
GenNm-SymPO
var1 -> len
1 Predictions for one round
var1
c
len
str
buf
2 Name candidates saved across rounds
3 Name validation following program data-flow
var1 -> c
4 Selected names
5 Name propagation following the call graph

Fig. 7: Inference pipeline of GenNm

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) [48] 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:

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

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 $n[f.\boldsymbol{fid}]$ denotes the map from a placeholder variable name to the ground-truth variable name for function f.

Context Propagation. Names in calling contexts can help the model understand the semantics of the function. Intuitively, names from the caller functions may provide hints about the higher-level purpose of the function, and names from the callee functions may provide details about the primitive functionalities of the analyzed function. 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 the contextual dataset during the training time 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:

$$\begin{aligned}\mathit{CallerCtx}(f, n) ::= \bigcup \big\{&\textbf{rename}(f.\boldsymbol{fid}(a), n[clr.\boldsymbol{fid}]) \\ &\big|(clr, f) \in b.\boldsymbol{cg} \land f.\boldsymbol{fid}(a) \in clr.\boldsymbol{body}\big\}\end{aligned} \tag{2}$$

$$\begin{aligned}\mathit{CalleeCtx}(f, n) ::= \bigcup \big\{&\textbf{rename}(cle.\boldsymbol{fid}(r), n[cle.\boldsymbol{fid}]) \\ &\big|(f, cle) \in b.\boldsymbol{cg} \land (r, \_) = cle.\boldsymbol{body}\big\}\end{aligned} \tag{3}$$

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

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 $\textbf{rename}(x, y)$ 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, $(clr, f) \in b.\boldsymbol{cg}$ describes the constraint that clr is a caller of f, and $f.\boldsymbol{fid}(a)$ refers to a call expression in the body of clr that calls f; $f.\boldsymbol{fid}$ denotes the placeholder name of f and a denotes the argument list. The propagation algorithm uses names in $n[clr.\boldsymbol{fid}]$ 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 $cle.\boldsymbol{fid}$ refers to the placeholder name of cle. Therefore, $cle.\boldsymbol{fid}(r)$ 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.

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:

$$\begin{aligned}D_{\text{ctx}} ::= \big\{&(\boldsymbol{query} : (f.\boldsymbol{body}, Ctx(f, n), f.\boldsymbol{ids}), \\ &\boldsymbol{resp} : n[f.\boldsymbol{fid}])\big|(b, n) \in \mathcal{D} \land f \in b\big\},\end{aligned} \tag{5}$$

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:

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

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. 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 [49]. 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.

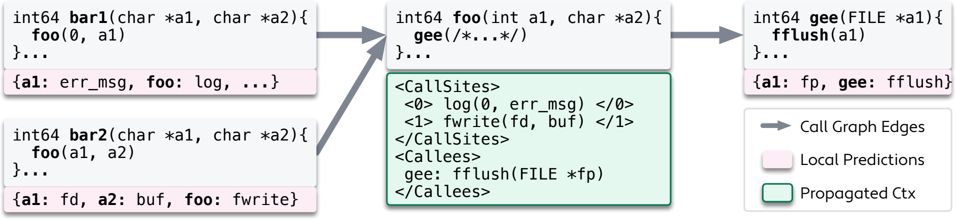
We first introduce how we construct the pairwise dataset used for SymPO (i.e., Step 2 in Fig. 6), and then introduce the SymPO loss (i.e., Step 3 in Fig. 6).

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 Section VI-G show that using ground-truth names underperforms compared to using the best predictions of GENNM$_{\text{Ctx}}$ as the preferred names. 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.

$$\begin{aligned}\hat{D} ::= \big\{&(\boldsymbol{query} : q,\ \boldsymbol{preds} : \hat{r},\ \boldsymbol{gt} : r)\big|(q, r) \in \mathcal{D}_{\text{loc}} \cup \mathcal{D}_{\text{ctx}} \\ &\land \hat{r} = \textsc{GenNm}_{\text{ctx}}(q, \text{top20})\big\},\end{aligned} \tag{7}$$



```
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. Initially, GENNM reasons each function independently and obtains the results shown in the pink boxes. After that, GENNM propagates names along the call graph. 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 function `gee()` with the predicted names, and propagates the renamed signature to the analyzed function. We can see that the names from caller functions hint the model that the purpose of `foo()` might be writing messages to a file, and the names from the callee function hint the model that `foo()` flushes the output buffer.

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

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

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

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. We use token-level precision and recall to measure the similarity between a predicted name and the ground-truth name.

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-G 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 a function if more than two-thirds of its callee functions do not have meaningful names. Optimizing model's preference on those data samples introduces only noises because the local information may not be enough for the model to predict good names. In addition, we remove functions with less than 5 statements and meanwhile do not have branches. Note that we only remove functions for constructing the SymPO training dataset. We do not remove any functions from the test dataset.

Loss function of SymPO. The loss function of SymPO is adapted from the loss function proposed in direct preference optimization [49], which is used to align human preference with fine-tuned LLMs. The loss function has two sub-goals: (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:

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

where Θ and Θ_{Ctx} denote the weights of $\text{GENNM}_{\text{SymPO}}$ and the weights of $\text{GENNM}_{\text{Ctx}}$, 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 $\text{GENNM}_{\text{Ctx}}$ 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 $\text{GENNM}_{\text{SymPO}}$, which is optimized by the loss function. It is initialized with the weights of $\text{GENNM}_{\text{Ctx}}$. The other model is a frozen $\text{GENNM}_{\text{Ctx}}$, which will not be updated during training. It is used as a "reference" model so that the divergence of $\text{GENNM}_{\text{SymPO}}$ is constrained. A detailed discussion for the loss function 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 and guides $\text{GENNM}_{\text{SymPO}}$ to produce a larger probability for b than $\text{GENNM}_{\text{Ctx}}$. Similarly, the red parts (corresponding to the second loss term) require $\text{GENNM}_{\text{SymPO}}$ to generate a significantly smaller probability compared to the $\text{GENNM}_{\text{Ctx}}$.

D. Context Augmentation at the Inference Stage

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 propagate names along the program call graph to provide contextual information. The algorithm (Step 5 in Fig. 7) is discussed in previous sections.

To select the final name prediction across different iterations, GENNM leverages program analysis to aggregate

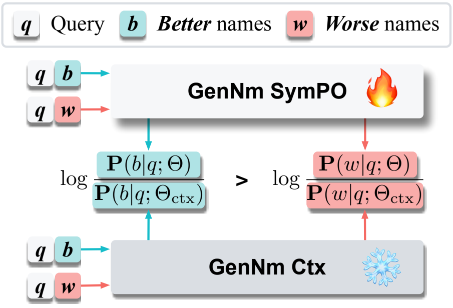

q Query
b Better names
w Worse names
q b
q w
GenNm SymPO
$\log \frac{\mathbf{P}(b|q;\Theta)}{\mathbf{P}(b|q;\Theta_{\text{ctx}})} > \log \frac{\mathbf{P}(w|q;\Theta)}{\mathbf{P}(w|q;\Theta_{\text{ctx}})}$
q b
q w
GenNm Ctx

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) ones are in red.

the names predicted in different iterations, and selects the candidate name with the maximum level of consistency. The analysis in GenNm is a general data-flow analysis customized to the domain of variable names, and the consistency check is achieved by majority voting. The implementation details of both techniques are in Appendix D of an extended version of this paper [66]. Note that there is existing work [62] exploring the combination of program analysis and LLM at the inference stage. Therefore, although our analysis is customized to the variable name domain and is different with existing work [62], we do not claim conceptual novelty for the analysis and voting algorithm.

V. Experimental Setup

A. Dataset

We evaluate GenNm on two commonly used [45], [35], [15] datasets. The first one is built following the same process as DIRTY [15] (noted as the DIRTY dataset) and the other is derived from the released VarCorpus dataset used by VarBERT [45] (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 [45]. 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 [45], 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 the same functions if they have the same name. We discuss the rationale in Appendix B. As a result, our processed datasets are more diversified than the existing datasets. For example, only 46% of functions in the original VarCorpus dataset have unique names, indicating that the other 54% of functions may have similar semantics (an example is in Fig. 15 in the Appendix). On the other hand, 81.3% and 89.4% of 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 [45], [15] use exact string match as the criterion for checking data leakage. However, as shown in Fig. 15 and Table VI 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 string similarity score to a training sample (from 0 to 100) is higher than 90.

Data availability. We submitted our artifact to the artifact evaluation track. We will publish our data splits, model checkpoints, and implementations upon publication.

B. Splits

For most experiments in the evaluation, 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 the placeholder name is already seen in the training data. To fairly compare the improvements achieved by GenNm with the baseline techniques, we additionally conduct experiments with the split-by-function setup following the previous work [45] in Section VI-A .

C. Models

Due to limited resources, we train GenNm from CodeGemma-2B [58] for most of the experiments. To study how different sizes of models may affect the performance, we additionally train two GenNm models from CodeLlama-7B and CodeLlama-34B [50] on the DIRTY dataset. The detailed hyperparameters of our model are listed in Table IX of an extended version of this paper [66]. We use VarBERT [45] and ReSym [62] as the baseline techniques. VarBERT [45] is a representative classification based method that demonstrates better performance than previous state-of-the-art models [35], [15]. ReSym [62] is a recent technique based on LLM. It also demonstrates better performance than previous state-of-the-art models [35], [15]. 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 [33], 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,

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

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

In Equations 10 and 11, $\simeq$ denotes whether two tokens have similar semantics. SymLM [33] built a semantics word cluster trained on CodeSearchNet [30] and derived edit-distance-based rules to measure the semantic 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 [56] on decompiled code summarization demonstrates that using GPT4 as an evaluator aligns better with human judgments than automatic metrics. Therefore, we adapt their method, further using GPT4 as an evaluator to measure the quality of generated names.

Specifically, we follow [56] 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. 19 of an extended version of this paper [66].

VI. Evaluation

A. Performance in terms of Semantics Match

Overall. We show the performance of GenNm compared with the baseline techniques in Table I. We can see that overall, GenNm outperforms both VarBERT and ReSym on all datasets/splits. On the DIRTY dataset, GenNm outperforms VarBERT by 8.5 percentage points in terms of both precision and recall; it outperforms ReSym by 5.6 and 4.6 percentage points in terms of precision and recall, respectively. On the VarCorpus dataset, GenNm outperforms VarBERT by 11.4 and 11.0 percentage points in terms of precision and recall, respectively; it outperforms ReSym by 9.5 and 6.2 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 overlaps between training and test sets with a stricter 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).
Project-in-train/project-not-in-train. Moreover, we observe that complex projects typically contain more than one binary. 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 [15]. As pointed out by previous work [45], 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 [22]) 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 all techniques perform better on samples whose projects have been 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, and it outperforms ReSym by more than 5 and 7 percentage points on the DIRTY dataset and the VarCorpus dataset, respectively. For the more challenging project-not-in-train samples, GenNm consistently outperforms both baseline techniques by 3.9–8.6 percentage points, demonstrating better generalizability.

TABLE I: Performance of GenNm compared with VarBERT and ReSym. 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 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
	ReSym	25.3	24.9	35.6	34.3	30.2	29.3
	GenNm	**30.5**	**28.8**	**41.7**	**39.6**	**35.8**	**33.9**
VarCorpus	VarBERT	20.9	19.3	32.5	31.0	29.8	28.3
	ReSym	23.5	24.1	34.2	35.8	31.7	33.1
	GenNm	**29.5**	**27.4**	**44.7**	**42.8**	**41.2**	**39.3**
VarCorpus Split by Function	VarBERT	-	-	-	-	50.0	49.2
	ReSym	-	-	-	-	51.2	52.2
	GenNm	-	-	-	-	**62.4**	**62.8**

TABLE II: Performance w.r.t. different sizes of base models.

Base Model	Proj. NIT		Proj. IT		Overall	
	PR	RC	PR	RC	PR	RC
CodeGemma-2B	29.7	28.0	38.5	36.7	33.7	32.0
CodeLlama-7B	29.9	28.8	36.7	35.5	33.1	31.9
CodeLlama-34B*	**35.9**	**33.4**	**39.5**	**37.4**	**37.1**	**35.3**

*We fine-tune CodeLlama-34B with LoRA.

Split by function. Moreover, following previous work [45], we further run all techniques on the VarCorpus dataset with the split-by-function setup. Split-by-function denotes the setup where some functions in a binary are in the training dataset while other functions are in the test dataset. We randomly sample 15% binaries from VarCorpus due to limited resources, following the practice of previous work [45], [15]. All techniques perform significantly better with the split-by-function setup. Especially, we can see that GenNm outperforms both baseline techniques by more than 10 percentage points. That is because the training paradigm and inference stage of GenNm enables it to leverage contextual information. In the split-by-function setup, the caller and callee functions of an analyzed function may already be seen during training. They provide higher quality contextual information than the caller/callee functions in the split-by-binary setup. Therefore, the performance of GenNm improves significantly. It demonstrates the effectiveness of leveraging calling contexts in the name recovery problem.

Significance of improvements. Note that the scale of improvement introduced by GenNm over the baselines is comparable to that in existing work. In the most challenging setup (split by binary, without overlap with training dataset), GenNm outperforms the baseline techniques by 4.6–11.4 percentage points. DIRTY [15] improves over its baseline by 5.1 percentage points (on the DIRTY dataset), and VarBERT [45] improves over its baseline by 4.5 percentage points (on the VarCorpus dataset). In the split-by-function setup, GenNm improves over the baseline techniques by 10.6–13.6 percentage points. VarBERT [45] improves over its baseline by 12.7 and 14.8 percentage points.

B. Performance w.r.t. Different Sizes of Base Models

GenNm fine-tunes pre-trained code language models. To study how base models with different sizes affect the performance, we additionally train GenNm with CodeLlama-7B and CodeLlama-34B [50]. Note that our resource cannot support a fully fine-tuning for the 34B model. Therefore, we use LoRA [28] to fine-tune the 34B model. We evaluate all models on a subset of the DIRTY dataset. The results are shown in Table II. We can see that GenNm fine-tuned from CodeLlama-34B achieves significantly better results than GenNm on CodeGemma-2B and CodeLlama-7B. Especially, for the most challenging setup where the project of a binary is not seen in the training dataset, the 34B version of GenNm outperforms the other versions by around 5 percentage points in both precision and recall. That demonstrates the training paradigm of GenNm can generalize to larger models.

C. Generalization to Different Compiler Optimizations

To evaluate the generalization of GenNm to other compiler optimization levels, we compare GenNm with both baseline techniques on programs compiled with different optimization levels from `-O0` to `-O3`. The results are shown in Fig. 10. We can see that GenNm outperforms both baselines across all optimization levels. It demonstrates that GenNm can generalize to optimized programs. The improvements of GenNm on programs compiled with less aggressive optimizations (i.e., `-O0` and `-O1`) are more significant than the improvements on programs compiled with `-O2` and `-O3`. That is because programs compiled with aggressive optimizations are significantly longer and diverge further from the distribution of source code. 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.

D. Generalization to Rare Names

We show GenNm generalizes better to rare names in Fig. 11. Observe that all techniques achieve better performance on names that appear more frequently in the training dataset, and GenNm consistently outperforms both baseline techniques on names with all name frequencies. Moreover,

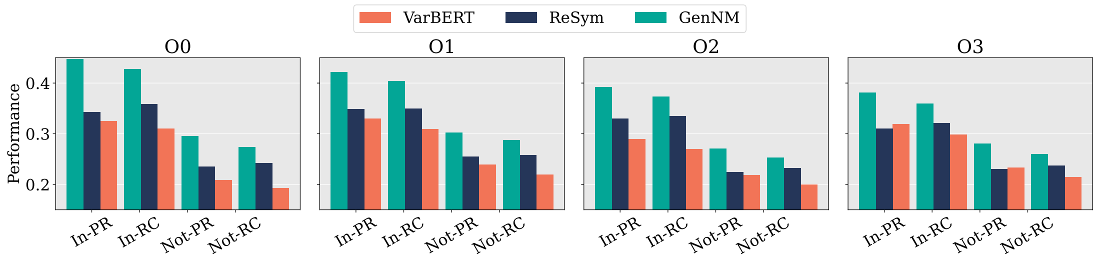


Fig. 10: Generalizability to other optimization levels. *In-PR*, *In-RC*, *Not-PR*, and *Not-RC* denote the average *precision* and *recall* on samples whose project is seen or not seen in the train data, respectively.

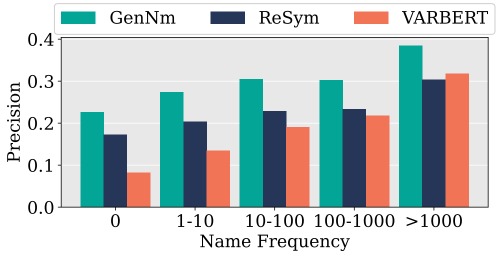


Fig. 11: Performance by name frequency on VarCorpus. The x-axis denotes the frequency of the ground-truth name for a variable in the training dataset of VarCorpus, and the y-axis the average precision achieved on the corresponding variables.

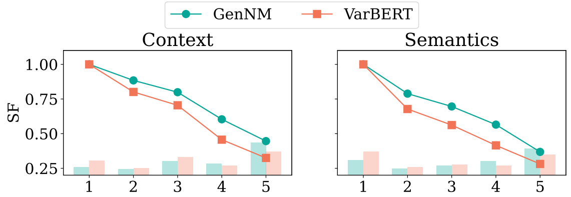


Fig. 12: Performance evaluated by the GPT4Evaluator. The two sub-figures show the scores for context relevance (Context) and semantics accuracy (Semantics), respectively. SF denotes the survival function. It indicates the number of samples achieving at least the corresponding score. The transparent bars reflect the distribution for each score.

GenNm is more robust when the frequencies of names decrease. For names that are never seen in the training dataset, both GenNm and ReSym outperform VarBERT. Especially, GenNm achieves a precision of over 20%, which is close to 2 times the performance of VarBERT on those variables. It supports that the generative model generalizes better than a classification model on unseen names.

The performance of GenNm on rare variables (i.e., variables with a name frequency from 1 to 10) is 27.1%, while the performance of VarBERT and ReSym are 13.5% and 20.4%, respectively. That indicates GenNm mitigates the biases of frequent names in the training dataset. Moreover, we show that 95% of the rare names are composed of frequently appeared tokens. Details are in Fig. 18 of an extended version of this paper [66].

TABLE III: 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
CodeLlama-70B	zero-shot	-	-
	3-shot	27.4	26.9
GenNm	-	**42.7**	**39.7**

E. 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. 12. 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% of 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. Fig. 19 in the Appendix shows examples for names with different scores. 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 most of the 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 of the semantics of the program, which is a challenging problem when the program does not have meaningful symbols [57]. We leave as future work to further improve the model's understanding of decompiled code.

F. Performance Compared to Blackbox LLMs

We compare the performance of GenNm with LLMs used as black-boxes. We randomly sample 1000 functions from the DIRTY dataset and query two state-of-the-art black-box LLMs (i.e., GPT-3.5 and GPT-4) and one large code LLM (i.e., CodeLlama-70B), with both the zero-shot and 3-shot setups. The prompts used are shown in Appendix H of an extended version of this paper [66]. The results are shown in Table III. Observe that GPT-4 achieves better performance than

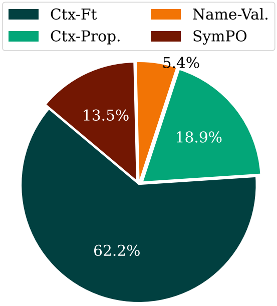


Fig. 13: Attribution of the improvements over ReSym to different components. *Ctx-Ft* and *SymPO* denote using the context-aware fine-tuning paradigm and the SymPO objective at the training stage, respectively. *Ctx-Prop.* and *Name-Val.* denote using the context propagation algorithm and the name validation algorithm at the inference stage, respectively.

GPT-3.5, and both LLMs achieve 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 underperform GENNM. GENNM outperforms the best results achieved by black-box LLMs by 11.3 and 6.4 percentage points in terms of precision and recall, respectively. For the code LLM, we observe most of its outputs have format errors in the zero-shot setup. We thus only calculate its performance on the 3-shot experiment. Note that it achieves results close to GPT-3.5 but inferior to GPT-4. We speculate that it is because the training data of GPT-3.5 and GPT-4 also contain significant amount of code. Therefore, the LLM specially trained on code may not have advantage given that its size is much smaller than GPT-4. It is worth noting that all LLMs are likely to be significantly larger in size (i.e., 10x–100x) than the base model used in GENNM. It demonstrates the necessity and effectiveness of fine-tuning a pre-trained code language model on this task.

G. Ablation Study

We conduct ablation studies to analyze how each component contributes to the effectiveness of GENNM. Moreover, we study the effectiveness of different design decisions in constructing the symbol preference dataset.

We run GENNM with different setups to study the effects of individual components. Recall that GENNM outperforms ReSym by 5.6% on the DIRTY dataset in terms of precision. In this study, we attribute the improvement to different underlying techniques. The results are in Fig 13. We can see that all components contribute to the improvements. Specifically, the context-aware fine-tuning paradigm (during training) and the context propagation algorithm (during inference) contribute most to the improvement, indicating the importance of leveraging contextual information. Moreover, we study how different degrees of contexts sensitivity in context propagation affect the performance of GENNM. The results show that a 5-degree context sensitivity empirically works well. Please see Appendix G of an extended version of this paper [66].

We further study how each design decision in constructing the symbol preference dataset affects the performance. The results are shown in Table IV. 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 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 in significantly worse performance than the default setup.

VII. CASE STUDIES

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. 19 of an extended version of this paper [66].

Malware reverse engineering. We use a real-world malware sample [39] to illustrate how GENNM helps a security analyst reverse engineer a malware sample. Fig. 14 shows a code snippet 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 [31]. 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 make 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 into 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 I of an extended version of this paper [66].

```
 v48 = recv(fd, v76, 0x1000, 0);
 v76[v48] = 0;
 for (i = strtok(v76, "\n");
   i && *i; i = strtok(0, "\n")) {
  //...`v52`:the last non-empty char in `i`
  if (*i == ':') {
   v53 = i;
   // parse the first part of the command
   while (1) {
    v54 = v53 - i;
    if (v52 <= v53 - i) break;
    v55 = *v53; v56 = v53++;
    if (v55 == 32) goto LABEL_98;
   }
   v56 = v53;
LABEL_98:
   *v56 = 0; v57 = v54;
   // stores the first part to `dest`
   strcpy(&dest, i + 1); strcpy(i, &i[v57 + 1]);
  }
 // find and execute the related command
 for (j = (const char **)&unk_60A500;
      *j; j = v66 + 2) {
  v73 = j;
  v65 = strcasecmp(*j, &s2);
  v66 = v73;
  if (!v65) {
   ((void(*)(int64, char *, char *))v73[1])
                            (fd, &dest, i);
}}}
```

(a) Decompiled code output by IDA

```
nbytes = recv(fd, buf, 0x1000, 0);
buf[nbytes] = 0;
for (tok = strtok(buf, "\n");
  tok && *tok; tok = strtok(0, "\n")) {
 //...`lastidx`:the last non-empty char in `tok`
 if (*tok == ':') {
  p1 = tok;
  // parse the first part of the command
  while (1) {
   i_len_0 = p1 - tok;
   if (lastidx <= p1 - tok)break;
   c = *p1; lastchar_0 = p1++;
   if (c == 32) goto LABEL_98;
 }
 lastchar_0 = p1;
LABEL_98:
 *lastchar_0 = 0; i_len_1 = i_len_0;
 // stores the first part to `dest`
 strcpy(&dest,tok+1);strcpy(tok,&tok[i_len_1 + 1]);
}
// find and execute the related command
for (cmd_ptr = (const char **)&unk_60A500;
       *cmd_ptr; cmd_ptr = cmd_tmp + 2) {
 matched_cmd = cmd_ptr;
 cmp = strcasecmp(*cmd_ptr, &cmd);
 cmd_tmp = matched_cmd;
 if (!cmp) {
   ((void(*)(int64, char *, char *))
        matched_cmd[1])(fd, &dest, tok);
}}}
```

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

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

TABLE IV: 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)

VIII. Related Work

Classification-based techniques for recovering symbol names. There are existing efforts on reconstructing variable names in stripped binary programs [15], [35], [45], [26], [62]. DEBIN [26] works on BAP-IR [9] that is more similar to the disassemble code than the decompiled code. It encodes facts in an IR program with probabilistic graph models(PGM) and predicts variable names based on the PGM. DIRTY [15] works on the decompiled code. It leverages a transformer model that interleaves predictions for variable names and variable types. State-of-the-art technique VarBERT [45] also leverages a transformer model working on the decompiled code. Different from DIRTY, which trains the model from scratch on the decompiled code, VarBERT first pre-trains the model on a large corpus of source code and then fine-tunes on the decompiled code. All three techniques formulate the problem as a classification task and thus can hardly predict names unseen in the training dataset. On the other hand, GenNm formulates it as a generative task. It can predict names that rarely appear in the training dataset.

Generative techniques for recovering symbol names. DIRE [35] is an early work leveraging a generative model (i.e., RNN) to solve the renaming problem. Yet it trains the RNN from scratch on a relatively small set of decompiled code and thus underperforms state-of-the-art techniques [45], [62] that can benefit from pre-training efforts on source code. ReSym [62] aims to recover names, types, and data structures. It shares a common goal with GenNm on name recovery. It fine-tunes an LLM for renaming variables, and uses program analysis as a post-processing step. In particular, it directly fine-tunes on individual decompiled functions using the ground truth type and name information. It further uses data-flow analysis to propagate type information and data structure fields, and leverages voting to resolve inconsistencies. GenNm goes beyond ReSym by proposing two unique ***training*** paradigms (i.e., Context-aware fine-tuning and SymPO) that more effectively train LLMs on the variable renaming problem. Although the program analysis components in both ReSym and GenNm are adapted from the standard data-flow analysis, they are different in both design and implementation. Specifically, the analysis in ReSym focuses on type inference and type checking, whereas the analysis in GenNm focuses on name propagation, which has a different nature.

Besides recovering variable names, another stream of work focuses on recovering function names in decompiled code [33], [34]. Their efforts are complementary to ours.

Reverse engineering. Existing efforts [67], [73], [64], [32], [57], [65] reverse engineer binary programs to analyze malware [55], [5], harden programs [19] and facilitate fuzzing [20], [16], [51]. Their efforts are complementary with ours, and the results of GenNm can benefit the reverse engineering tasks, as shown in Section VII.

Foundational binary program analysis. GenNm relies on existing foundational binary analysis techniques [3], [7], [38], [10] to process binary programs, such as disassembly [42], [72], type recovery [53], [36], [54], [71], [73], and decompilation [69]. State-of-the-art achieves good performance in most cases [47], [6], [70].

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 design context-aware fine-tuning to teach the model how to leverage contextual information, and design symbol preference optimization to mitigate models' biases. Our prototype GenNm demonstrates significant improvements on SOTA in challenging setups.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. We are grateful to the Center for AI Safety for providing computational resources. This research was supported in part by DARPA VSPELLS - HR001120S0058, IARPA TrojAI W911NF-19-S-0012, NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

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] 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."

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

[8] E. Bogomolov, A. Eliseeva, T. Galimzyanov, E. Glukhov, A. Shapkin, M. Tigina, Y. Golubev, A. Kovrigin, A. van Deursen, M. Izadi *et al.*, "Long code arena: a set of benchmarks for long-context code models," *arXiv preprint arXiv:2406.11612*, 2024.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 463–469.

[10] 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.

[11] 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.

[12] 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.

[13] 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

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[15] 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.

[16] 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.

[17] (2024) Cve-2018-4407. https://github.com/github/securitylab/tree/main/SecurityExploits/apple/darwin-xnu/icmp_error_CVE-2018-4407.

[18] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[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) Groups mitre att&ck. https://attack.mitre.org/groups/.

[23] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[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] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[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] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2021.

[29] 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.

[30] 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.

[31] (2023) A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro/.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] (2024) Virustotal. https://www.virustotal.com/gui/file/03cfe768a8b4ffbe0bb0fdef986389dc.

[40] 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.

[41] 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.

[42] 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.

[43] OpenAI, “Gpt-4 technical report,” 2023.

[44] 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

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

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

[56] 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.

[57] 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.

[58] C. Team, “Codegemma: Open code models based on gemma,” *arXiv preprint arXiv:2406.11409*, 2024.

[59] 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.

[60] 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

[61] 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

[62] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, “Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries,” in *2024 ACM SIGSAC Conference on Computer and Communications Security*.

[63] X. Xu, S. Feng, Y. Ye, G. Shen, Z. Su, S. Cheng, G. Tao, Q. Shi, Z. Zhang, and X. Zhang, “Improving binary code similarity transformer models by semantics-driven instruction deemphasis,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1106–1118.

[64] X. Xu, Z. Xuan, S. Feng, S. Cheng, Y. Ye, Q. Shi, G. Tao, L. Yu, Z. Zhang, and X. Zhang, “Pem: Representing binary program semantics for similarity analysis via a probabilistic execution model,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 401–412.

[65] 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.

[66] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan, and X. Zhang, “Symbol preference aware generative models for recovering variable names from stripped binary,” 2024. [Online]. Available: https://arxiv.org/abs/2306.02546

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 659–676.

[73] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.

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 decreasing the probability for worse names.

On the other hand, the first term constrains 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 quality consideration. (2) We only include executable binary programs in our dataset, precluding intermediate relocatable binary files since the semantics of a relocatable file rely on its symbol table [60], 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 keeps other binaries as simple wrapper programs. Take the tool Bibutils [3] as an example. The (corresponding source code files of) two binary programs `xml2ris` [4] and `xml2end` [5] 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 V, both our processed datasets are smaller than the original VarCorpus dataset, while their diversity is 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 are present in the training dataset). Previous work [45], [15], [35] considers two functions as the same only when their normalized strings are exactly the same. However, it may overestimate 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. 15 to illustrate the problem. All three versions try to allocate some memory and terminate the execution on failure. Versions (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. Suppose that version (a) is in the training dataset. The performance of a model on version (b) may not reflect the generalizability of the model. On the other hand, the third function has semantic 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 [6] as the metric to conservatively check potential data leakage. The string similarity is based on string edit distance, ranging from 0 (indicating two strings have no overlap) to 100 (indicating two strings are an exact match). Empirically, we consider a test sample as overlapped with a training data sample if their string similarity is larger than 90.

Table VI 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 is substantially better than the performance shown in Table I. The performance, unfortunately, cannot faithfully reflect the capability of the models on the variable recovery problem.

C. Correlation between `memset` and `buffer`

We observe that a model is more likely to predict a variable name as `buffer` if the variable is used as the first parameter of `memset`. To quantify our observation, we use Chi-2 test [7] to test the correlation between "a variable is the first parameter of `memset`" and "a variable is predicted the name `buffer`". The null hypothesis is that the distribution of the two random variables are independent. As shown in Table VIII, the results of Chi-2 test reject the null hypothesis

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

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

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

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

[7] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chisquare.html

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

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

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

TABLE V: 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**

```
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. 15: 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.

TABLE VI: 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**

TABLE VII: Correlation between the model's predictions and the corresponding function names. The first column denotes whether a variable is the first parameter of memset, columns 2–3 and 4–5 whether a variable is named as buffer or file, respectively. The last row shows the Chi-2 p-values for memset and buffer, and memset and file, respectively. A smaller value denotes higher correlation.

memset	buffer		file	
	T	F	T	F
T	19	700	2	717
F	292	206504	165	206631
χ^2	1.6e-63		0.22	

with a p-value significantly smaller than 1e-5 (i.e., 1.6e-63), indicating that the two random variables are indeed correlated with a statistical significance. In other words, "a variable is the first parameter of memset" is indeed correlated with "a variable is predicted the name buffer". In comparison, we also run the same test with memset and another randomly picked name file. The Chi-2 test yields a p-value of 0.22, not supporting the correlation between memset and file.