Beyond Classification: Inferring Function Names in Stripped Binaries via Domain Adapted LLMs

Linxi Jiang*
The Ohio State University
jiang.3002@osu.edu

Xin Jin*
The Ohio State University
jin.967@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

***Abstract*—Function name inference in stripped binaries is an important yet challenging task for many security applications, such as malware analysis and vulnerability discovery, due to the need to grasp binary code semantics amidst diverse instruction sets, architectures, compiler optimizations, and obfuscations. While machine learning has made significant progress in this field, existing methods often struggle with unseen data, constrained by their reliance on a limited vocabulary-based classification approach. In this paper, we present SYMGEN, a novel framework employing an autoregressive generation paradigm powered by domain-adapted generative large language models (LLMs) for enhanced binary code interpretation. We have evaluated SYMGEN on a dataset comprising 2,237,915 binary functions across four architectures (x86-64, x86-32, ARM, MIPS) with four levels of optimizations (O0-O3) where it surpasses the state-of-the-art with up to 409.3%, 553.5%, and 489.4% advancement in precision, recall, and F1 score, respectively, showing superior effectiveness and generalizability. Our ablation and case studies also demonstrate the significant performance boosts achieved by our design, *e.g.*, the domain adaptation approach, alongside showcasing SYMGEN's practicality in analyzing real-world binaries, *e.g.*, obfuscated binaries and malware executables.**

I. INTRODUCTION

Inferring function names for stripped binaries is crucial for many binary analysis and reverse engineering tasks, such as malware detection [45], vulnerability discovery [39], [71], decompilation [5], and code similarity detection [54]. Binary function names act as a succinct abstraction of the semantics of functions embedded within the binary code. Through this abstraction, security professionals can efficiently comprehend the core semantics of functions, eliminating the need for a comprehensive and time-consuming reverse engineering process. For example, Mandiant, a recognized cybersecurity company, applies function name inference in binary malware analysis by automatically annotating function names in assembly code to better understand malicious behavior and develop effective countermeasures [67].

Yet, function name inference for commercial-off-the-shelf (COTS) binaries presents substantial challenges. COTS binaries are compiled from varied compilation settings that encompass different computer architectures and optimization levels, thereby producing a variety of binary code. Complicating matters further, COTS binaries originate from source projects developed by individuals with diverse backgrounds and for a wide range of purposes, such as IoT firmware or malware [7]. This diversity in development origins introduces additional variability in the binary code and its contextual semantics [41]. Furthermore, a significant hurdle in function name recovery for COTS binaries is the binary stripping process, which results in the maximal semantic loss (*e.g.*, removing all the symbol names, except those used for dynamic linking, including function names, variable names, and debugging symbols if the program is compiled with these information). COTS binaries are also potentially fortified by protective measures, *e.g.*, obfuscation, impeding reverse engineering of binary code semantics.

Encouragingly, recent research [31], [52], [20] has illuminated a promising direction in leveraging machine learning and language models to decode binary code semantics. However, these approaches often fall short of achieving a truly *generalizable* solution. Instead, they frequently yield models that tend to be overfitted on their training data but show poor performance on unknown data. For example, while SYMLM [31] demonstrates promising performance, achieving an F1 score over 0.73 on its own test set, its effectiveness significantly diminishes when applied to previously unseen binaries, *e.g.*, test functions excluded from the training dataset, with performance decreasing to a mere 0.08 F1 score. This stark decline in performance underlines the models' poor generalizability, rendering them less effective for practical application on real-world COTS binaries. Moreover, a critical limitation of these approaches is their reliance on weak base models, such as LSTM [79] for NFRE [20], and BERT [17] for SYMLM [31] and XFL [52]. These models introduce high variance, particularly when they conceptualize the function name prediction task as a classification problem, inherently limiting their generalizability. The classification-based solutions confine the predictive model to selecting names and words from their training vocabularies, which represent only a small segment of the broader spectrum of natural language semantics.

Meanwhile, substantial advances are witnessed in the realm of machine learning for constructing generalizable models (*e.g.*, ChatGPT and Llama) that have achieved state-of-the-art performance in numerous tasks, such as question answering and natural language understanding [74], [32]. A notable strategy underpinning the success of these models is the adoption of generative large language models (LLMs) that benefit from task-agnostic pre-training on extensive natural language corpora. Remarkably, these generative LLMs have shown promising results in zero- and one-shot learning scenarios, demonstrating significant performance on tasks previously unseen during training, with

* The first two authors contributed equally to this work.

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.2025.240797
www.ndss-symposium.org

minimal or no task-specific data. Despite these advances, a critical limitation emerges when applying generative LLMs to the domain of binary code understanding: the inability to accurately grasp the "meaning" of binary code due to a lack of domain-specific knowledge [28]. The training datasets for these LLMs predominantly consist of natural language text with a small proportion of source code data [66], [56], devoid of the specialized content necessary to comprehend binary code semantics.

In this paper, we present SYMGEN, the first generative framework designed to infer function names for stripped binaries in a robust and generalizable manner. Unlike existing classification-based research, SYMGEN conceptualizes the function name prediction problem as an autoregressive generation task. This paradigm shift allows for a more subtle and flexible generation of function names, catering to the inherent complexities of binary code. To overcome the generalizability challenges, SYMGEN harnesses the capabilities of generative LLMs. Recognizing the conventional LLMs' shortfall in binary code comprehension, we employ a domain adaptation strategy specifically tailored to the binary code summarization task. This approach is crucial in bridging the knowledge gap, thereby equipping the LLMs with the required insight into binary code semantics. Furthermore, SYMGEN incorporates decompiler output, *i.e.*, decompiled code, for addressing the variability in binary code resulting from diverse compilation settings. Confronting the limitations posed by resource constraints—particularly the impracticality of full parameter training for LLMs—we adopt a parameter-efficient learning methodology within SYMGEN. This approach utilizes LoRA-based matrix decomposition [25] to optimize the training process, significantly reducing the computational resources required while maintaining the model's performance and scalability.

Our comprehensive evaluations reveal SYMGEN's exceptional generalizability and performance superiority over current state-of-the-art solutions across four architectures (x86-64, x86-32, ARM, MIPS) and four optimization levels (O0-O3). Specifically, SYMGEN demonstrated remarkable improvements in precision, recall, and F1 score—outperforming the state-of-the-art by 409.3%, 553.5%, and 489.4%, respectively. Additionally, our tests on real-world binaries, including obfuscated binaries, IoT firmware images, and malware executables, have demonstrated SYMGEN's security practicality. For instance, when evaluated against obfuscated binaries, SYMGEN outperforms the state-of-the-art by an astonishing 488% increase in F1 score. SYMGEN can also successfully predict malware's function names, e.g., `get_random_ip`, assisting malware analysis.

We have also evaluated the effectiveness of SYMGEN's components by ablation studies. First, when using decompiled code as input, SYMGEN achieves a 285.7% higher F1 score compared to using assembly code as input. Additionally, we find that domain adaptation with LLM-generated summaries improves the performance with increases of 6.36% in F1 score. With the parameter-efficient learning approach, the computational resource usage is significantly reduced, *e.g.*, the training duration is condensed to 37 hours on a single A100 GPU, which is considerably shorter than full-parameter training that takes 21 days on 2,048 A100 GPUs [66]. SYMGEN's advanced performance with our design and use of individual components further confirms its effectiveness and novelty.

Contributions. We make the following contributions:

- We propose a novel framework for binary function name inference, leveraging LLMs to achieve unprecedented levels of generalizability and accuracy.
- We present an advanced domain adaptation approach and a parameter-efficient learning strategy for tuning the pretrained generative LLMs to binary semantic modeling.
- Our work outperforms existing solutions in function name prediction, setting new generalizability benchmarks, with practical utility to obfuscated binary analysis. Our dataset and code are publicly available at https://github.com/OSUSecLab/SymGen.

II. BACKGROUND AND RELATED WORK

A. Preliminary

Inferring function names from stripped binaries is a process of reverse engineering the semantics of binary functions and then generating accurate function names that reflect this function semantics. Unlike existing works [20], [31], [23] that define the problem as a classification problem, *i.e.*, selecting function name words from training vocabularies as the predicted function names, we define this problem as a *generation task*.

Formally, given the input binary function f ($f \in F$) that can be tokenized into a sequence of binary code tokens, *i.e.*, $f = \{t_1, t_2, ..., t_n \mid t_i \in \mathcal{V}_f\}$, the function name inference model $\mathcal{M}$ ($\mathcal{M} : F \rightarrow S$) generates an output sequence of natural language words $s = \{w_1, w_2, ..., w_m \mid w_j \in \mathcal{V}_s\}$ ($s \in S$). In this paper, the model $\mathcal{M}$ (*i.e.*, SYMGEN) follows an autoregressive generation paradigm, *i.e.*, $p(s) = \prod_{j=1}^{m} p(w_j \mid w_1, ..., w_{j-1}, f)$, where the generation of each word w_i depends on both the binary function semantics and the preceding words ($\{w_1, ..., w_{i-1}\}$). This problem formulation is critical for boosting the *generalizability* of SYMGEN. Here, we define the model's generalizability as its performance on previously unseen functions that are not included in its training set.

B. Binary Reverse Engineering

In software development, binary stripping is commonly used to remove meaningful debugging symbols from binaries, including function names and type information. This process, while beneficial for performance and security, leads to a remarkable reduction in semantic information within the binaries, posing significant challenges for reverse engineers in analyzing the underlying code semantics. To mitigate these challenges, many disassembly and decompilation techniques [4], [68] have been developed to bridge the gap created by the absence of these crucial symbols.

Specifically, disassembly translates binary code—consisting of machine-level code—back into assembly code, and the resulting assembly code is a format more intelligible to humans. Existing binary function name prediction works [31], [52], [20], [14], [33] predominantly rely on the assembly code obtained by disassembling binaries. For example, SYMLM predicts function names from assembly code along with a trace-based model [31]. ASMDEPICTOR encodes binary function semantics from assembly code using a Transformer model [52]. Decompilation goes a step further. It recovers the missing semantic information and generates a roughly equivalent source code or pseudo-code, which is typically in a form resembling C, albeit approximated.

Recently, we have also observed an increasing trend of using machine learning for symbol recovery tasks, *e.g.*, function

name prediction [14], [20], [31], [33], [52], [51] and variable name/type inference [23], [10], [48], [53], [78], which could enhance decompiler output. For example, DIRTY [10] models the local decompiled code semantics for predicting variable names and types. NERO [14] grasps the function semantics using convolutional neural networks. Note that, *function name prediction is different from the other binary reverse engineering tasks, e.g., variable and type inference, as it requires modeling the global semantics of functions*.

C. Generative LLMs

Generative LLMs are models that produce output based on input sequences, following the autoregressive generation paradigm, *i.e.*, the generation of tokens depends on the previously generated tokens [66]. These models can be categorized into encoder-decoder models and decoder-only models based on model architectures [74], [29]. The encoder-decoder model, adopting the Transformer architecture, involves two separate components for processing input and output [57]. The encoder processes the input data and encodes it into a context-rich representation, while the decoder generates output based on this representation. The decoder-only models, on the other hand, are based solely on the Transformer decoder module [56], [66]. They are particularly known for their effectiveness in understanding and generating texts. The GPT model family (including the well-known ChatGPT and GPT-4) by OpenAI is a prime example of a decoder-only model [56], [47], [1]. The decoder-only model architecture, while simpler compared to encoder-decoder models, proves effectiveness in a wide range of tasks [62], [74]. More importantly, decoder-only models have exhibited substantial capabilities in terms of generalizability under the zero- or few-shot setting [6], distinguishing them from other generative language models. In addition to the generative LLMs, there are also language models that leverage the encoder-only model architecture, *e.g.*, BERT [17]. These models are specifically designed for encoding input, but they are not primarily geared towards generating text. In function name prediction, existing LLM-based solutions mainly adopt the encoder-only model [31], [52].

III. Overview

A. Challenges

C1: Generalizable Function Name Inference. The generalizability of function name inference models is very important in dealing with COTS binaries in real-world reverse engineering tasks. Yet, these binaries, compiled from a variety of source projects infused with domain-specific knowledge from multiple fields, present a significant challenge in learning their semantics. This complexity is compounded by the fact that COTS binaries are compiled from source projects crafted by developers with distinct backgrounds and intended for various production purposes. Additionally, COTS binaries, commonly stripped, carry very little semantic information which makes learning and reverse engineering their semantic representations especially hard. Furthermore, the diversity of binary code is also mirrored in the output of the models; that is, the range of function names that need to be predicted is also varied [31].

C2: Knowledge Gaps between Natural Language Tasks and Function Name Inference. While generative LLMs, such as ChatGPT and Llama, exhibit promising potential for our task, their efficacy in comprehending binary code is prohibitively poor [28]. This limitation significantly impedes their direct application in inferring function names [28]. Unlike the objectives of many generative tasks, such as question answering where the models excel, function names are highly abstract and encapsulate the core semantics of functions in a very succinct manner. Consequently, the substantial knowledge gaps between the natural language processing capabilities of generative LLMs and the specific requirements for interpreting binary code in our task present a formidable challenge.

C3: Resource-expensive Training of LLMs. The advanced performance of LLMs presents a promising direction for function name inference. However, this potential is tempered by the extraordinarily high costs associated with hardware resources, time, and data availability required for model training. The prevalent practice of full-parameter training demands substantial resources; for instance, training the Llama model necessitates the use of 2048 NVIDIA A100 GPUs and 21 days [66]. Additionally, there is an absence of sufficiently large binary datasets that can satisfy the requirements for training these LLMs. The creation of a high-quality binary dataset with accurate ground truth annotations entails significant human efforts. Consequently, employing LLMs for our task poses notable challenges in model training.

C4: Data Leakage and Duplication. Data leakage—test binary functions also appear in the training set—significantly affects the accuracy of evaluating generalizability. Additionally, the training set may contain popular but duplicated binary functions, leading to model overfitting on these functions and reducing performance on new, unseen samples [9], [37]. Moreover, using well-trained models as a base for training can also introduce leakage through their pretraining process. Detecting this leakage and identifying duplicates is extremely challenging for several reasons. First, stripped binary functions compiled from the same source code can appear different due to factors like varying binary symbol addresses (introducing unique address-specific callee function names). Second, the pretraining datasets for these well-trained models are typically closed-source, making it difficult to track potential overlaps.

B. Key Insights

S1: Autoregressive Function Name Inference with Generative LLMs. Existing generative LLMs demonstrate exceptional generalizability, attributable to their autoregressive training paradigm. In contrast, the base models for prior function name prediction efforts are predominantly trained using the masked language modeling objective, employing encoder-only models pre-trained in this manner. This approach allows the model to consider both preceding and succeeding contexts during prediction, but such a setting does not hold for real-world applications because the succeeding context is missing. In contrast, autoregressive training and inference accurately mirror the nature of real-world generation processes, thereby offering enhanced generalizability [6]. Consequently, we apply autoregressive training to SymGen.

S2: Domain-adaptive Learning with Function Summaries. To address **C2**, a straightforward method involves directly finetuning the model on a binary function name inference dataset. This method, however, faces limitations due to the inherent abstractness of function names, which encapsulate only a fraction of the function's semantics. And different function

```
1  /* Return the address of the last file name component of
   ↪  NAME.  If NAME has no relative file name components
   ↪  because it is a file system root, return the empty
   ↪  string.  */
2
3  char * last_component (char const *name)
4  {
5      char const *base = name + FILE_SYSTEM_PREFIX_LEN
       ↪  (name);
6      char const *p;
7      bool last_was_slash = false;
8
9      while (ISSLASH (*base))
10         base++;
11
12     for (p = base; *p; p++)
13     {
14         if (ISSLASH (*p))
15             last_was_slash = true;
16         else if (last_was_slash)
17         {
18             base = p;
19             last_was_slash = false;
20         }
21     }
22
23     return (char *) base;
24 }
```

Figure 1: Function Summary of `last_component` from GNU Sed Project

names may emphasize various aspects of a function's semantics, leading to inconsistencies in semantic representation. For example, Figure 1 presents the `last_component` function, in which the function summary (at line 1) describes retrieving the last component of a *file path*—a detail not explicitly captured by its name. This discrepancy underscores that function summaries, typically presented as comments, are rich in semantic information, detailing function behavior in natural language, thereby playing to the strengths of LLMs. Such summaries can serve as a crucial bridge to narrow the knowledge between binary function semantics and the descriptive richness required for accurate interpretation.

S3: Parameter-efficient Learning. In addressing **C3**, we leverage a parameter-efficient training methodology customized to our target generative LLMs. Using the fine-tuning optimization [16], [25], we freeze the weights of the pre-trained model and employ a gradient decomposition technique to produce low-rank representations of these weights. These low-rank weights significantly reduce the parameter count compared to the original model weights, thus enhancing computational efficiency. Notably, the attention and feed-forward layers account for a substantial portion of the model's size and demand considerable resources for updates. Consequently, we specifically apply parameter reduction and integrate the resultant low-rank weights into the model via residual connections, optimizing performance without losing the model's original capacities (*e.g.*, the instruction following ability).

S4: Data Deduplication and Leakage Mitigation. For **C4**, we have proposed systematic methods to reduce data duplication and leakage. We start by defining the leakage and duplication problems from two angles: in-dataset and cross-dataset duplication. With this framework, we identify duplicates by (*i*) matching the same source function names or function bodies during compilation, and (*ii*) detecting duplicated but syntax-different binary functions compiled from the same source functions using callee normalization. To address leakage in the base model, we apply the prevalent membership inference technique to identify potential leakage in its pretraining.

C. System Overview

The workflow of SYMGEN consists of three major steps:

(I) Domain adaptation by function summarization involves the strategic training of SYMGEN's base model to enhance its knowledge of binary code using function summaries.

(II) Parameter-efficient learning minimizes the number of training parameters for both domain adaptation and function name inference tasks by layer weight selection, weight freezing, matrix decomposition, and residual connections.

(III) Finetuning for function name generation transfers the knowledge from the domain-adapted model, as developed in Step (I), to the task of inferring function names.

IV. DETAILED DESIGN

A. Domain Adaptation

Our objective is to develop a *generalizable* model capable of learning binary code semantics and generating accurate names for binary functions. To this end, we exploit the capabilities of pre-trained generative LLMs to grasp the complex and dynamic semantics of binary code as described in §III-B. To leverage these models for binary function name inference, the intuitive methods include zero-shot (or few-shot) prompting and direct finetuning for our task, where finetuning usually can boost the models' performance more. However, as revealed through our evaluations in §VI-D, the straightforward fine-tuning achieves the desired efficacy, with the latter resulting in a performance that is 8.21% inferior to our proposed approach. Our analysis indicates that this is due to the models' limited exposure to binary code during training, as they are predominantly developed on natural language corpora [6], [66]. Although these corpora may include a minor set of source code (*e.g.*, 5% in Llama [66]), source code is fundamentally different from binary code, where source code is with rich semantics, *e.g.*, function and variable names, data types, and structural details. This difference leads to a significant knowledge gap between the models' pre-training on natural language (and to some extent, source code) and the specific requirements of our binary code comprehension task, thereby resulting in the observed performance inefficiencies. To mitigate this knowledge gap of LLMs, we propose to perform domain adaptation by training the models on summarizing decompiled code.

Figure 2 presents the workflow of SYMGEN's domain adaptation approach. Our initial step involves curating open-source C projects, which have been extensively utilized in prior learning-based binary analysis research [48], [31], [52]. These projects are then compiled into binaries, with debugging symbols to facilitate analysis across diverse computer architectures and optimization levels. Subsequent to binary stripping, the generated stripped binaries are decompiled into decompiled code. To adapt the LLMs on binary code comprehension, we need to obtain the dataset with ground truth, *i.e.* the description of binary code semantics. For this, we initially sought to align with methodologies from source code summarization research [2], which typically uses function comments as ground truth. However, we observe that the noise within these function comments poses a significant challenge,

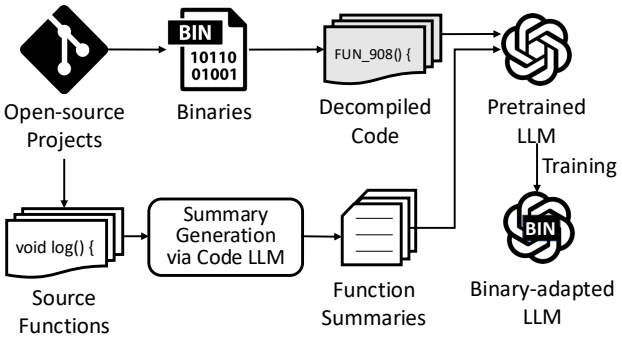


Figure 2: Domain Adaptation by Function Summarization

potentially compromising SYMGEN. In this paper, we propose two approaches to address this challenge, including (1) function comment preprocessing and noise removal, and (2) summary generation using code LLMs.

Function Comment Preprocessing and Noise Removal. To understand the noises in function comments, we first perform a manual study on the function comments of 900 randomly sampled source functions from 45 open-source projects. Overall, we have identified five categories of noises, stemming from varying coding conventions and assumptions, as well as the often ad-hoc nature of development processes and practices. For example, one category of errors is the comment used for communication, instead of summarizing the function semantics, where developers use keywords like "TODO" and "FIXME" to communicate with others. More details of these noises are provided in Appendix §A for interested readers.

In order to remove the noises from the function comments and form a dataset with high-quality ground truth, we have followed the noise-removal solutions from existing code comment cleaning research [61]. Specifically, we run their open-sourced artifact[1] on our source code dataset. This process involves two steps: (1) identifying the source functions and their associated function comments using abstract syntax tree (AST) parsing, and (2) cleaning these comments using the rule-based artifact [61]. For step (1), we have provided more details of our engineering efforts in Appendix §B. Note that, for function comments, we consider the comment, right above the function signature, as our target and directly extract it from source files for each function.

Summary Generation via Code LLMs. In addition to the approach of function comment preprocessing, we have also explored the use of LLMs for generating high-quality function summaries for domain adaptation. Our design choice is motivated by recent advancements in code LLMs, such as Code Llama [60] and StarCoder [38], which demonstrate impressive capabilities in generating succinct, high-quality source code summaries. Moreover, it has been found that the output generated by these powerful LLMs can serve as high-quality references for training other LLMs [72]. For example, ShareGPT is a set of dialogue corpus, generated by ChatGPT, which has been widely used for training other LLMs [13], [18]. Therefore, we opt to use Code LLMs to generate summaries for binary functions via understanding their source code.

[1] https://github.com/BuiltOntheRock/FSE22_BuiltOntheRock

While code LLMs can produce seemingly accurate comments, a major concern is their potential for hallucination, *i.e.*, generating summaries that express incorrect or nonexistent semantics. To address this issue, we observe that LLMs can self-improve, *i.e.*, they can correct their own errors via reasoning [27]. Therefore, we have adopted techniques from prior studies of self-consistency checking [72], [12], [42]. Specifically, we employ the universal self-consistency method [12], a straightforward yet effective approach to eliminate hallucinations. We query the LLM multiple times to produce several summary candidates. We then use the LLMs again to evaluate these candidates and select the most consistent summary among them as the final output. Through a systematic series of experiments involving various prompts, we identify the most effective prompt for our task, as shown in Figure 9a (in Appendix §C). In this process, we select and use the Code Llama model, which is fine-tuned from the Llama 2 model, for its state-of-the-art performance on code understanding [60].

To this end, we have proposed two approaches to address the long-standing issue of noise in function comments [77], [11], to the best of our efforts. Our subsequent evaluations indicate that (1) both approaches can boost SYMGEN's effectiveness, and (2) the summary generation-based approach is more effective for SYMGEN, *e.g.*, SYMGEN with the summary generation-based approach has 5.81% better precision than SYMGEN with preprocessed comments (see §VI-D). Consequently, we have selected summary generation as our default solution for domain adaptation. With the decompiled code and generated ground-truth summaries of binary functions, we proceed to train LLMs for domain adaptation using a parameter-efficient learning approach.

B. Parameter-efficient Learning

Existing binary function name prediction models, *e.g.*, SYMLM [31] and XFL [52], commonly involve full-parameter supervised finetuning. This method essentially fits models to suit a specific binary code dataset. However, this fine-tuning process updates all parameters of the model, demanding significant computational resources, *e.g.*, it takes 8 days for SYMLM's fine-tuning as reported by the paper [31]. In SYMGEN, we leverage more advanced generative LLMs, which have significantly more parameters than the base models of SYMLM and XFL, *i.e.*, BERT [17]. Therefore, this inefficiency leads to substantial challenges in training resources and time. To address this issue, numerous efficient finetuning efforts have been proposed. For example, parameter-frozen tuning integrates tunable low-rank matrices into model layers and has shown superior performance compared to full-parameter finetuning [69], [25]. Observing these advances, we leverage a parameter-efficient learning framework based on LoRA [25], as shown in Figure 3. In particular, we focus on the attention and feed-forward layers of SYMGEN and perform low-rank matrix decomposition and residual connection on the frozen layer weights.

Layer Weight Selection. SYMGEN is built upon the generative LLM using the transformer decoder model architecture. This model architecture consists of four major layers, including (i) masked multi-head attention layers, (ii) multi-head attention layers, (iii) add & norm layers, and (iv) feed-forward layers. Different from BitFit [76] and DiffPruning [21], which selectively adjust a subset of the model's layers, SYMGEN's training focuses on layers with trainable weights. That is, we

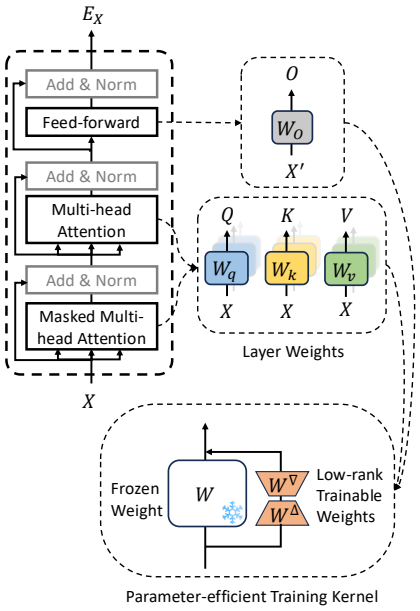


Figure 3: Parameter-Efficient Learning. In the Transformer decoder model, we train the low-rank weights for the multi-head attention layers and feed-forward layers.

exclude the add & norm layers, which inherently have no trainable parameters. This design choice of tuning all learnable weights is to optimize SYMGEN to better understand binary code semantics. For this, we focus on SYMGEN's attention layers (*i.e.*, masked multi-head attention layers and multi-head attention layers) and feed-forward layers. One of the shared features of these layers is that they have the same mechanism of performing dot products of layer weights and the input. Formally, given a sequence of decompiled code tokens $X = \{x_1, x_2, ..., x_n\}$, the attention layer produces the key, value, and query matrices, *i.e.* K, V, and Q by:

$$k_i = W_k \cdot x_i, \quad q_i = W_q \cdot x_i, \quad v_i = W_v \cdot x_i \tag{1}$$

where $k_i \in K$, $q_i \in Q$ and $v_i \in V$ for $1 \leq i \leq n$. $W_k \in \mathbb{R}^{d \times d}$, $W_q \in \mathbb{R}^{d \times d}$, and $W_v \in \mathbb{R}^{d \times d}$ are learnable weight matrices. The feed-forward layer essentially encapsulates dot product calculation, *i.e.*, $O = W_o \cdot X'$. In this paper, we focus on tuning the weights of W_k, W_q, W_v, and W_o as illustrated in Figure 3.

Low-rank Weight Decomposition. In deep neural networks, the weight updates are performed by the backward propagation process, in which the gradients of layer weights are calculated to update themselves [35]. For parameter-efficient training, we use the low-rank matrix decomposition approach proposed by existing parameter-frozen training research [25], [16]. As shown in Figure 3, we first freeze the original model weights W ($W \in \mathbb{R}^{d \times d}$ can be any of W_k, W_q, W_v, and W_o). Freezing the model weights can not only reduce the computational resources required for training but also preserve the instruction-following capacities of our target pretrained models. Next, SYMGEN creates the low-dimension trainable weight by decomposing each frozen weight into two low-rank matrices ($W^\nabla \in \mathbb{R}^{d \times r}$ and $W^\Delta \in \mathbb{R}^{r \times d}$) which have far less parameters compared with the original weights. Specifically, given the frozen weight W, the backward propagation generates the gradient $\delta W \in \mathbb{R}^{d \times d}$. SYMGEN decomposes this gradient into W^∇ and W^Δ, so that $\delta W = W^\nabla \cdot W^\Delta$. In these low-rank weights, r is far less than d, *i.e.*, $r \ll d$, which decreases the number of parameters that need to be trained from $d \times d$ to $2 \times r \times d$. This decomposition can dramatically reduce the trainable weights (*e.g.*, our training on the 13.5 GB Llama model only requires 20.3 MB weight updates). Eventually, the low-rank matrices are integrated into attention and feed-forward layers by residual connections [24].

C. Finetuning for Function Name Inference

SYMGEN aims to accurately infer function names for stripped binaries. To achieve this, we finetune the LLM that has been adapted to the binary code understanding domain (§IV-A) on binary function name inference dataset.

Input Masking. As discussed in §III-B, SYMGEN takes as input the decompiled code from stripped binaries and generates function names. This decompiled code carries the artificial function names generated by decompilers, such as `FUN_080021e` where `080021e` is the function address. We observe that simply feeding the model with the raw decompiled code can hurt its performance since the model to be finetuned can be misled by the artificial function names. One simple solution is to add instructions to let the model avoid focusing on these artificial names but this method cannot effectively address the problem due to two reasons. First, artificial function names vary as the function addresses change. Second, the function body can also include artificial function names for callees. Considering these noises, we formalize the fine-tuning task as a cloze-solving problem [63], *i.e.*, we mimic the process of how human beings learn the semantics of words by the context. Similarly, in SYMGEN, binary function names are inferred and generated by understanding the context information of the function semantics. To achieve this, the artificial function name in decompiled code is replaced by the `[MASK]` token without altering any other portion of the decompiled code (*e.g.*, function body and function parameters). It is worth noting that the `[MASK]` token is widely used in pretraining LLMs. The resultant decompiled code is used as the input to SYMGEN. For readers' interest, Figure 9c (in Appendix §C) presents SYMGEN's prompt for completing the cloze-solving task.

Finetuning Ground Truth and Objective. For finetuning, we have used the developer-created function names as ground truth, following the common practice as existing research [31], [52], [20], [14]. Specifically, we obtain it from the DWARF entries of the binaries with debugging symbols since we have full control of the dataset generation process. To process the ground truth names into tokens of the SYMGEN's base model vocabulary, we use the BPE tokenizer[2] downloaded along with our base model, *i.e.* Code Llama. To train the model on our binary function name generation dataset, we formalize the finetuning process as the autoregressive generation task. Formally, for a tokenized input sequence $\mathbf{x} = \{x_1, x_2, x_3, ..., x_n\}$ and output sequence $\mathbf{y} = \{y_1, y_2, y_3, ..., y_m\}$, SYMGEN aims to maximize the likelihood following the forward autoregressive factorization model:

$$\max_\theta \log P_\theta(\mathbf{y}) = \sum_{i=1}^{m} \log P_\theta(y_i \mid \mathbf{x}; y_{1:i-1}) \tag{2}$$

[2]We use Huggingface tokenizers that comes with SYMGEN's base model.

$$\log P_\theta(y_i \mid \mathbf{x}; y_{1:i-1}) = \frac{\exp(h_\theta(\mathbf{x}; y_{1:i-1})^\top e(y_i))}{\sum_{y'} \exp(h_\theta(\mathbf{x}; y_{1:i-1})^\top e(y'))} \quad (3)$$

where $y_{1:i-1}$ denotes the preceding tokens of y_i. $h_\theta(\cdot)$ is the hidden states produced by SYMGEN's intermediate layers for context representation. $e(\cdot)$ is the input embedding.

V. SYMGEN DATASET AND LEAKAGE MITIGATION

We collect 33 open-sourced C projects from GNU Software and widely used libraries, including those that have been extensively used by existing binary reverse engineering research [31], [20], [52], *e.g.*, `OpenSSL`, `binutils` and `bash` (the detailed list of projects is presented in Table VI of Appendix §D).

A. Dataset Construction

Function Name Prediction Dataset. We compile above source projects into four computer architectures (x86-32, x86-64, ARM, and MIPS) and four optimization levels (O0, O1, O2, and O3) using GCC-9.4.0. Our dataset contains 9,842 unique binaries and 2,237,915 binary functions in total. We randomly split our dataset into training, validation, and test sets with the ratio of 8:1:1 at the binary level, same as prior works [14], [31]. Note that, the binary compilation renders both binaries with and without debugging information. To construct datasets for our task, we obtain the input decompiled code from stripped binaries and the ground truth function names by analyzing the DWARF entries from binaries with debugging information.

Domain Adaptation Dataset. Using compiled binaries, we also construct our domain adaptation datasets by the two approaches proposed in §IV-A, *i.e.*, (1) function comment preprocessing and noise removal, and (2) summary generation via code LLMs. After applying our proposed steps to obtain the function comments and LLM-generated summaries, we observe a common problem of matching source function comments/summaries and binary functions, because the comments and summaries are obtained at the source level, not from binaries. To address this issue, we connect source function comments/summaries and stripped binary functions by establishing connections using binary function addresses. Specifically, after obtaining the function comments or summaries (§IV-A), we first connect the source and binary functions by matching source function names and the binary function names from the DWARF entries of binary with debugging information. Next, we further parse the DWARF entries to connect the source function names and the binary function start and end addresses. As the start and end addresses of a function remain unchanged regardless of whether it is stripped, we link stripped and unstripped binary functions by matched function addresses. To this end, we successfully match the function comments/summaries with its corresponding stripped binary functions.

B. Data Leakage Mitigation and Deduplication

In real-world binaries, there are functions that inevatably used across projects for reasons such as code reuse and the use of libraries, *e.g.*, `binutils` binaries share a large amount of code that is copied. We observe that the datasets of existing works, *e.g.*, SYMLM [31], commonly contain many duplicated functions in the test set which also occurs in the training set. In machine learning evaluations, these duplicated samples are defined as data leakage [22], *i.e.*, the duplicated functions have been leaked to the model during training. While facing this issue, we observe that existing function name inference works do not fully resolve it. In particular, SYMLM [31] and XFL [52] do not consider data leakage in their design. ASMDEPICTOR [33] removes duplicates by matching the input code exactly, which unfortunately fails to identify duplicate samples that differ slightly. For example, Figure 4 presents code snippets of two functions from ASMDEPICTOR's training and test sets, respectively. These two functions originate from the same source function but from different binaries, resulting in distinct symbol addresses. Consequently, the highlighted addresses in lines 1 and 3 differ between these two functions. Unfortunately, ASMDEPICTOR ignores their equivalence due to these superficial differences.

```
...
mov_esi_ 0x40c78c ,
mov_rdi_rax,
call_ 0x2e24 ,
mov_qword_ptr_[rbp-0x10]_rax,
cmp_qword_ptr_[rbp-0x10]_0,
...
```

(a) ASMDEPICTOR's Training Sample (Line 37,492 in its Training File)

```
...
mov_esi_ 0x41ce4c ,
mov_rdi_rax,
call_ 0xfffffffffffb75b ,
mov_qword_ptr_[rbp-0x10]_rax,
cmp_qword_ptr_[rbp-0x10]_0,
...
```

(b) ASMDEPICTOR's Test Sample (Line 27 in its Test File)

Figure 4: Data Leakage in ASMDEPICTOR. The two samples, compiled from the same source function but in different function addresses, are treated as different samples.

Dataset Duplicates and Leakage. In this paper, we have observed two types of dataset duplicates: (1) cross-dataset duplicates and (2) in-dataset duplicates. Cross-dataset duplicates are defined as binary functions that appear in both the training and test sets, which is commonly defined as data leakage. In-dataset duplicates refer to functions that appear more than once within the training set or the test set, such as `syncok` and `copy_termtype` showing several times in our training set. For the purpose of generalizability evaluation, it necessitates removing the leaked samples, *i.e.*, cross-dataset duplicates, from our test sets. However, the question of whether deduplicating in-dataset duplicates within the training set (or test set) is necessary remains unanswered. Such a question was unfortunately ignored by prior research and they keep in-dataset duplicates in both training sets and test sets. In this paper, we have conducted ablation studies to validate the performance of both SYMGEN and our baselines with and without in-dataset duplicates in §VI-D. Our rigorous evaluations show that removing in-dataset duplicates from the training sets can effectively address overfitting and improve model performance by 2.94×. Therefore, unless otherwise specified, we remove in-dataset duplicates in our subsequent evaluations by default.

Data Deduplication and Leakage Mitigation. Based on these observations, we employ a more rigorous deduplication

Algorithm 1: Callee Normalization in Stripped Decompiled Code

```
1  Function visit(node):
2      switch node.type do
3          case expression:
4              visitExpression(node)
5          ...
6  initialize funcMap := {}
7  Function visitPostfixExpression(node):
8      if node.primaryExpression exists and node.LeftParen
         exists then
9          callFunName := node.primaryExpression.Identifier
10         if callFunName in funcMap then
11             callFunName := funcMap[callFunName]
12         else
13             newFunName := "FUN_" + funcMap.count
14             funcMap[callFunName] := newFunName
15             callFunName := newFunName
16     return visit(node.children);
```

approach to mitigate in-dataset and cross-dataset duplicates. To be more thorough, we perform deduplication on two levels: (*i*) duplicates with the same function names or binary code, and (*ii*) duplicates with the same source function but different binary code due to different addresses, as shown in Figure 4. For level (*i*), we remove binary functions that either share the same source function names or have different names but identical function bodies. For level (*ii*), our approach is motivated by the observation of examples in Figure 4. Specifically, the decompiled code of stripped binaries often contains address-related callee function names in its call statement. These addresses in callee names can appear different due to variations in the memory layout of different binaries, resulting in duplicated decompiled functions with distinct function bodies. To avoid duplicates that AsmDepictor failed to identify, we propose algorithm 1 to normalize callee function names for deduplication. Specifically, we first parse the decompiled binary functions from stripped binaries to generate their abstract syntax trees (ASTs). Next, as shown in lines 1 to 5, we perform a depth-first search on the generated AST to traverse all nodes. Finally, we normalize the callee function names by substituting them with placeholders using the `visitPostfixExpression` function (lines 6 to 16). An example of decompiled code before and after applying algorithm 1 is provided in Figure 10a and Figure 10b (in Appendix §E). It is worth noting that our approach might aggressively filter out false positive cases, *e.g.*, functions having the same names but different semantics. However, we argue that such a rigorous approach will result in a reliable dataset to accurately reflect SymGen's generalizability. Eventually, our approach identifies 77.9% duplicated functions (*i.e.*, 1,742,421 binary functions). The resultant dataset contains 495,494 unique binary functions, which is comparable in size to the dataset of existing research [52].

Leakage Detection in SymGen's Base Model. SymGen is trained based on the Code Llama model [60], and the samples of our test set can be potentially leaked in Code Llama's training process. However, its pertaining dataset is unfortunately closed-source, posing significant challenges in detecting this leakage. Although Code Llama is mostly trained on source code, which is significantly different from the stripped binary code in our test set, we argue it is crucial to detect such leakage to ensure the quality of our dataset. For this, we have applied a membership inference approach, prevalent in previous dataset inference work [8], [40]. Specifically, we prompt our base model to complete the incomplete stripped binary code of our test sets and measure its completion effectiveness. Following the common setting [8], [40], we prepare the incomplete stripped binary code by removing the last 5 lines of each decompiled function while the average number of lines is 32. We use the popular metrics, *i.e.*, exact matching and CodeBLEU [58], as the completion metric. The evaluations of our test datasets on Code Llama present scores of 0 in exact matching, indicating that none of the test samples are completed. Additionally, the average CodeBLEU similarity score is only 0.098, where CodeBLEU is computed by n-gram matching that matches the variable names and symbols, and we find that generated lines only match some symbols without producing complete lines. To this end, we argue that our test sets are not leaked to SymGen's base model in its training process.

VI. Evaluation

We have implemented SymGen with 3,068 lines of own code using the pretrained Code Llama with its own BPE tokenizer[3] as the base model for SymGen, which is code-specific version of Llama 2 and finetuned on six programming languages, *e.g.*, C++ [60]. We employ Ghidra [44] to parse binaries and generate decompiler output. It is important to underscore that SymGen's design is agnostic to decompilers. We opt to use Ghidra due to its open-source nature and its widespread recognition and adoption. We develop the source and decompiled code parsers based on ANTLR 4 [49]. For parameter-efficient learning and finetuning, we have developed the training pipeline on top of Pytorch [50], transformers [73], alpaca-lora [65], and loralib [26].

Our evaluations aim to answer the following research questions:

- **RQ1**: What is SymGen's overall performance, *i.e.*, across different optimization levels and architectures?
- **RQ2**: Compared with the state-of-the-art, how effective and generalizable is SymGen?
- **RQ3**: How can SymGen's components and strategic design, *e.g.*, domain adaptation, improve its performance?
- **RQ4**: Can SymGen predict function names for real-world binaries, *e.g.*, obfuscated binaries and malware executables?

A. Experimental Setup

Evaluation Environment. Our evaluations are performed on a server equipped with a 48-core AMD EPYC 7643 processor at 2.3 GHz, 921 GB memory, and four NVIDIA A100 GPUs with 80 GB VRAM each, running RHEL 8.6 OS.

Evaluation Metrics. We adopt the evaluation metrics, *i.e.*, precision, recall, and F1 score, that are commonly used by prior works [31], [20], [14]. Given the ground truth function name S and inferred function name $\hat{S}$, we split them into individual

[3]The model is downloaded from Huggingface: https://huggingface.co/codellama

words, *i.e.*, $S = \{w_1, w_2, ..., w_n\}$ and $\hat{S} = \{\hat{w_1}, \hat{w_2}, ..., \hat{w_n}\}$ and then calculate precision, recall and F1 score by:

$$TP = \sum_{\hat{w_i} \in \hat{S}} \mathbb{1}_S(\hat{w_i}), FP = \sum_{\hat{w_i} \in \hat{S}} 1 - \mathbb{1}_S(\hat{w_i}), FN = \|S\| - TP$$

$$\mathbb{1}_S(\hat{w_i}) = \begin{cases} 1 & \text{if } \hat{w_i} \in S \\ 0 & \text{if } \hat{w_i} \notin S \end{cases}$$

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

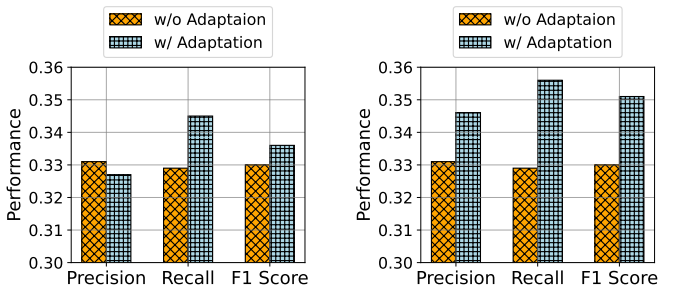
where TP, FP, and FN are true positive, false positive, and false negative. We adopt the CodeWordNet of SYMLM [31] to address label noise introduced by semantically similar words. More illustration of our evaluation metrics is provided in Appendix §F for readers of interest.

Baselines. In this paper, we choose SYMLM, ASMDEPICTOR [33], and XFL [52] as our baselines. For these three baselines, we train their models using our dataset under the same settings as SYMGEN to ensure a fair comparison. According to the XFL paper, it is only applicable to x86-64 binaries due to their design with VEX IR [52]. Therefore, we evaluate XFL and compare it with SYMGEN on x86-64 binary dataset. We have tried different training parameters for our baselines and selected the best performance following their training instructions. We skip evaluating NERO [14], NFRE [20], and DEBIN [23] as our baselines have demonstrated superior performance.

B. RQ1: Overall Performance of SYMGEN

To answer **RQ1**, we have evaluated SYMGEN on our dataset. Table I presents its performance across the four computer architectures and four optimization levels. For each architecture and optimization level, we train an individual model, separately, and test it on our corresponding test set. Overall, SYMGEN achieves a precision of 0.275, a recall of 0.281, and an F1 score of 0.277 on average. For the x86-64 architecture, SYMGEN achieves an average precision of 0.373, a recall of 0.388, and an F1 score of 0.380. On the x86-32 binaries, SYMGEN records a precision of 0.324, a recall of 0.338, and an F1 score of 0.330. On MIPS and ARM, SYMGEN's performance yields precision scores of 0.136 and 0.265, recall scores of 0.138 and 0.260, and F1 scores of 0.137 and 0.262, respectively. In this performance, we have observed the performance discrepancy of SYMGEN between x86-64/x86-32 and ARM/MIPS binaries, which is the same as those of SYMLM. For instance, SYMGEN exhibits 1.45× better performance on x86-64 binaries compared to ARM binaries, whereas SYMLM shows a 1.37× improvement under similar conditions. We hypothesize that this difference stems from the architecture-specific application binary interfaces (ABIs), which introduce more challenges in understanding MIPS/ARM binaries.

SYMGEN demonstrates optimal average performance on binaries optimized at the O3 level, achieving an F1 score of 0.290. Notably, SYMGEN performs better on binaries with higher optimization levels (O2 and O3) compared to those with lower levels (O0 and O1), with the exception of MIPS binaries. This improved performance can be attributed to the higher optimized binaries containing clearer function semantics and fewer invalid instructions, which aligns with observations made by Theodoridis *et al.* [64].



(a) Preprocessed Comments **(b) LLM-generated Summaries**

Figure 5: SYMGEN Performance with and without Domain Adaptation Using (a) Preprocessed Comments and (b) LLM-generated Summaries

C. RQ2: Baseline Comparison

To deepen our analysis, we also evaluate our baselines on our dataset, and the evaluation results are reported in Table I. Generally, ASMDEPICTOR obtains the performance of 0.054 precision, 0.043 recall, and 0.047 F1 score on average. In contrast, SYMLM achieves the average precision of 0.114, recall of 0.099, and 0.105 F1 score across all computer architectures and optimization levels. XFL achieves an average precision of 0.189, recall of 0.156, and F1 score of 0.171.

More importantly, we observe the significant superiority of SYMGEN over all baselines, achieving up to 1650%, 1280%, and 1455.6% improvements in precision, recall, and F1 score, respectively. Specifically, SYMGEN is more effective than SYMLM, and its average precision, recall, and f1 score across all computer architectures and optimization levels are 241.2%, 283.8%, and 263.8% times these of SYMLM, respectively. Additionally, it outperforms ASMDEPICTOR with 409.3%, 553.5%, and 489.4% improvements on average precision, recall, and F1 score, respectively. For XFL, although it outperforms ASMDEPICTOR and SYMLM on x86-64 binaries, SYMGEN also achieves 197%, 249%, and 222% better precision, recall, and F1 score, respectively, compared to XFL. From the results, we observe that the performance metrics of baselines on our dataset are not as high as those reported in their respective papers. For instance, SYMLM achieved an average F1 score of 0.73 on its own dataset, which drops to 0.105 on our dataset. The F1 score of XFL also drops from 0.68 on the DEBIN dataset to 0.16 on our deduplicated dataset. As discussed in §V-B, this discrepancy is largely due to data leakage issues in their test sets, which do not accurately reflect real-world performance when duplicates are effectively removed, as is the case with our dataset. This observation is consistent with findings from the unknown binary evaluation in the SYMLM paper [31], where both SYMLM achieved F1 scores lower than 0.10 on unknown samples. ASMDEPICTOR also mentioned function deduplication in paper [33]. Similarly, XFL's performance drops significantly on unseen function names as presented in its paper [52]. That said, SYMGEN has dramatically advanced the state-of-the-art, showing much better effectiveness and generalizability.

Table I: Overall Performance of SYMGEN and Our Baselines. Note that ↑ and ↓ indicate the improvement and deterioration of SYMGEN over baselines.

Arch	Model	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
			O0			O1			O2			O3	
x86-64	SYMGEN	0.346	0.356	0.351	0.376	0.387	0.381	0.383	0.401	0.392	0.388	0.407	0.397
	SYMLM	0.099	0.100	0.100	0.101	0.103	0.102	0.106	0.092	0.098	0.109	0.094	0.101
	ASMDEPICTOR	0.059	0.043	0.050	0.067	0.055	0.060	0.061	0.045	0.052	0.075	0.060	0.067
	XFL *	0.169	0.140	0.153	0.188	0.133	0.156	0.196	0.177	0.186	0.203	0.175	0.188
	SYMGEN v.s. ASMDEPICTOR	(↑486.4%)	(↑727.9%)	(↑602.0%)	(↑461.2%)	(↑603.6%)	(↑535.0%)	(↑527.9%)	(↑791.1%)	(↑653.8%)	(↑417.3%)	(↑578.3%)	(↑492.5%)
	SYMGEN v.s. SYMLM	(↑249.5%)	(↑256.0%)	(↑251.0%)	(↑272.3%)	(↑275.7%)	(↑273.5%)	(↑261.3%)	(↑335.9%)	(↑300.0%)	(↑256.0%)	(↑333.0%)	(↑293.1%)
	SYMGEN v.s. XFL	(↑104.7%)	(↑154.3%)	(↑129.4%)	(↑100.0%)	(↑191.0%)	(↑144.2%)	(↑95.4%)	(↑126.6%)	(↑110.8%)	(↑91.1%)	(↑132.6%)	(↑111.2%)
x86-32	SYMGEN	0.297	0.315	0.306	0.314	0.324	0.319	0.342	0.356	0.348	0.342	0.358	0.349
	SYMLM	0.098	0.114	0.106	0.138	0.106	0.120	0.111	0.133	0.121	0.085	0.112	0.096
	ASMDEPICTOR	0.059	0.045	0.051	0.061	0.033	0.043	0.057	0.050	0.053	0.057	0.040	0.047
	SYMGEN v.s. ASMDEPICTOR	(↑403.4%)	(↑600.0%)	(↑500.0%)	(↑414.8%)	(↑881.8%)	(↑641.9%)	(↑500.0%)	(↑612.0%)	(↑556.6%)	(↑500.0%)	(↑795.0%)	(↑642.6%)
	SYMGEN v.s. SYMLM	(↑203.1%)	(↑176.3%)	(↑188.7%)	(↑127.5%)	(↑205.7%)	(↑165.8%)	(↑208.1%)	(↑167.7%)	(↑187.6%)	(↑302.4%)	(↑219.6%)	(↑263.5%)
MIPS	SYMGEN	0.144	0.148	0.146	0.138	0.140	0.139	0.130	0.131	0.130	0.135	0.134	0.134
	SYMLM	0.124	0.133	0.146	0.118	0.155	0.139	0.150	0.085	0.108	0.130	0.071	0.091
	ASMDEPICTOR	0.056	0.044	0.049	0.078	0.059	0.067	0.050	0.040	0.045	0.044	0.036	0.040
	SYMGEN v.s. ASMDEPICTOR	(↑157.1%)	(↑236.4%)	(↑198.0%)	(↑76.9%)	(↑137.3%)	(↑107.5%)	(↑160.0%)	(↑227.5%)	(↑188.9%)	(↑206.8%)	(↑272.2%)	(↑235.0%)
	SYMGEN v.s. SYMLM	(↑16.1%)	(↑11.3%)	(↑0.0%)	(↑16.9%)	(↓16.7%)	(↑0.0%)	(↓15.4%)	(↑54.1%)	(↑20.4%)	(↑3.8%)	(↑88.7%)	(↑47.3%)
ARM	SYMGEN	0.240	0.238	0.239	0.265	0.257	0.261	0.268	0.269	0.269	0.280	0.276	0.280
	SYMLM	0.088	0.042	0.057	0.113	0.061	0.079	0.135	0.097	0.113	0.122	0.089	0.103
	ASMDEPICTOR	0.044	0.044	0.044	0.049	0.037	0.042	0.030	0.033	0.032	0.016	0.020	0.018
	SYMGEN v.s. ASMDEPICTOR	(↑445.5%)	(↑466.7%)	(↑443.2%)	(↑440.8%)	(↑594.6%)	(↑521.4%)	(↑793.3%)	(↑715.2%)	(↑740.6%)	(↑1650.0%)	(↑1280.0%)	(↑1455.6%)
	SYMGEN v.s. SYMLM	(↑172.7%)	(↑440.9%)	(↑319.3%)	(↑134.5%)	(↑321.3%)	(↑230.4%)	(↑98.5%)	(↑177.3%)	(↑138.1%)	(↑129.5%)	(↑210.1%)	(↑171.8%)

* For XFL, its design and implementations are limited to x86_64 binaries, as reported by its paper [52]. Therefore, XFL is only evaluated on our x86_64 dataset.

D. RQ3: Ablation Study

Domain Adaptation by Function Summarization. The purpose of domain adaptation is to improve SYMGEN's capacities of binary code comprehension by mitigating the knowledge gap of our target pretrained LLMs. To validate its effectiveness, we perform the comparative evaluations on SYMGEN by studying its performance with and without domain adaptation (1) the function comments that have been preprocessed with noise removal and (2) the function summaries generated by LLMs. For SYMGEN without domain adaptation, we directly perform finetuning on function name inference with parameter-efficient learning.

We first present the results of domain adaptation using our preprocessed function comments in Figure 5a. This domain adaptation strategy enhances SYMGEN's recall and F1 score by 4.86% and 1.82% respectively, but it results in a decline in precision by 1.21%. In contrast, as shown in Figure 5b, domain adaptation with LLM-generated summaries improves SYMGEN across all metrics, with increases of 4.53%, 8.21%, and 6.36% in precision, recall, and F1 score, respectively. Through cross-comparison, we observe that LLM-generated summaries are more effective than function comments, enhancing precision by 5.81%, recall by 3.19%, and the F1 score by 4.46%. Overall, these findings indicate that domain adaptation can significantly enhance SYMGEN's performance in function name prediction. Furthermore, we also note that SYMGEN without domain adaptation tends to generate meaningless function names, such as `FUN_876458`. After performing domain adaptation, such meaningless function names are significantly reduced by 71.7% in inference results.

Parameter-efficient Learning. Parameter-efficient learning significantly reduces both the training duration and GPU memory demands for SYMGEN. For context, the full-parameter pretraining of the Llama model necessitates the use of 2,048 A100 GPUs, each equipped with 80 GB of VRAM, spanning approximately 21 days—an aggregate of 163,840 GB of GPU VRAM [66]. Moreover, finetuning the Llama model is recommended to have at least 8 A100 GPUs, cumulatively amounting to 640 GB of VRAM [59]. In contrast, leveraging our parameter-efficient learning approach enables the training of SYMGEN with the Code Llama 34B model on only one A100 GPU within a considerably reduced timeframe of around 37 hours. This stark reduction in resource utilization underscores the efficiency of our method, making it a viable solution for environments with limited hardware capabilities. In addition, we also observe that parameter-efficient learning can effectively assist SYMGEN to converge. While setting the total number of epochs as 16, we observe the model training converges at about 6 epochs. Figure 11 (in Appendix §G) presents the training loss of SYMGEN across the training epochs. This early stop can effectively avoid overfitting of SYMGEN on our training dataset, particularly in the context of the supervised learning process.

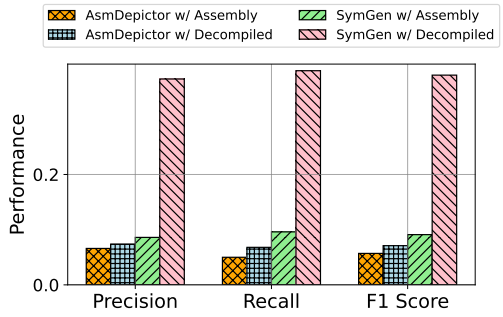


Figure 6: Performance of SYMGEN and ASMDEPICTOR Using Decompiled Code and Assembly Code as Input

Decompiled Code as Binary Representations. Given the superior performance of SYMGEN over our baselines, we examine our design choice of using decompiled code as model

Table II: Performance of SYMGEN and Baselines on Training Set with and without Duplicates

Model	Training Set	Precision	Recall	F1 Score
ASMDEPICTOR	w/ duplicates	0.019	0.015	0.017
	w/o duplicates	0.059	0.043	0.050
SYMLM	w/ duplicates	0.117	0.068	0.086
	w/o duplicates	0.099	0.100	0.100
XFL	w/ duplicates	0.159	0.144	0.151
	w/o duplicates	0.169	0.140	0.153
SYMGEN	w/ duplicates	0.353	0.329	0.341
	w/o duplicates	0.346	0.356	0.351

Table III: Performance of SYMGEN and Baselines on Test Set with and without Duplicates

Model	Test Set	Precision	Recall	F1 Score
ASMDEPICTOR	w/ duplicates	0.057	0.042	0.048
	w/o duplicates	0.059	0.043	0.050
SYMLM	w/ duplicates	0.103	0.093	0.097
	w/o duplicates	0.099	0.100	0.100
XFL	w/ duplicates	0.152	0.138	0.144
	w/o duplicates	0.169	0.140	0.153
SYMGEN	w/ duplicates	0.348	0.369	0.358
	w/o duplicates	0.346	0.356	0.351

input, by comparing it to the use of assembly code as input. For cross-comparison, we have also enhanced our baseline models by retraining them on our decompiled code dataset and using the same experimental setup, described in §VI-A. Here, we choose ASMDEPICTOR as the baseline for this ablation study due to several reasons. First, we could not adapt SYMLM to decompiled code since it was originally trained on a microtrace-based model with specific input representations designed for assembly code [31], which cannot be applied to decompiled code. XFL's binary analysis process is based on VEX IR, which cannot be adapted to decompiled code as well. Consequently, we focus on ASMDEPICTOR and retrain it using the exact same configurations outlined in its original paper. We only modify its input to decompiled code, with all other settings unchanged. For comparison, we updated SYMGEN's domain adaptation and fine-tuning processes to account for nuances specific to handling assembly code as input. This involves modifying the prompts used during finetuning to align with assembly code, *e.g.*, the prompt for finetuning is shown in Figure 9b (in Appendix §C).

Figure 6 presents the evaluation results. For ASMDEPICTOR, the decompiled code improves its precision, recall, and F1 score by 12.1%, 36.0%, and 24.6%. In contrast, with decompiled code serving as the binary function representation, SYMGEN achieves 333.7% improvement in precision, 304.2% in recall, and 317.6% in F1 score over the assembly code input variant. Moreover, for the same decompiled input, SYMGEN is also much more effective than ASMDEPICTOR, *i.e.*, with the 404.1%, 470.6%, and 435.2% better precision, recall, and F1 score, respectively. Considering our objective on generalizability, we attribute the outperformance of SYMGEN to its unique problem formalization as well as the efficient and generalizable training design.

In-dataset Deduplication. To investigate whether in-dataset duplicates, which were defined in §V-B, in the training set can affect model performance, we also train our baselines and SYMGEN using training sets with and without such duplicates. Table II presents the evaluation results. Overall, we observe that these duplicates significantly reduce the performance of our baselines, while they have negligible impact on SYMGEN. For instance, the F1 scores of SYMLM, XFL, and ASMDEPICTOR without training set duplicates are 1.23×, 1.01×, and 2.94× better compared with duplicates. Meanwhile, SYMGEN achieves an F1 score of 0.341 with duplicates in the training set, which is 2.6% lower than the performance without duplicates. While we could have compared SYMGEN with baselines on the with-duplicate setting to show better outperformance, we chose to deduplicate training sets to give a fair comparison and report baselines' better performance. In-dataset duplicates can cause the model to overfit on the training set and memorize these repeated data rather than learning generalizable features. This results in poor performance when the model encounters unseen data. Such overfitting is also widely noted by prior research [3], [34], [9], [37].

In addition to training sets, we have also evaluated the in-dataset duplicate removal for test sets. Table III presents the performance of SYMGEN and baselines on the duplicated test set, where the F1 scores of ASMDEPICTOR, SYMLM and XFL are 0.048, 0.097, and 0.144, respectively. In contrast, their F1 scores without test set duplicates are 0.050, 0.100, and 0.153, showing minor differences of only 3.09%, 4.0%, and 5.8%. Similarly, SYMGEN achieves an F1 score of 0.358 on the duplicated test set, compared to 0.351 on the non-duplicated test set. This indicates that in the duplicated test set, despite some function names appearing more frequently than others, the final performance of the models is not significantly impacted.

Table IV: Performance of SYMGEN, SYMLM, and XFL on Obfuscated Binaries

Obfuscation	SYMGEN	SYMLM	XFL
w/o obfuscation	0.302	0.071	0.139
bcfobf	0.288 (-4.6%)	0.063 (-11.3%)	0.044 (-215.9%)
cffobf	0.271 (-10.3%)	0.056 (-21.1%)	0.034 (-308.8%)
subobf	0.303 (+0.3%)	0.062 (-12.7%)	0.039 (-256.4%)

E. RQ4: Effectiveness on Real-world Binaries

Obfuscated Binaries. Binary Obfuscation is widely used to increase the complexity of the code, making binary analysis and reverse engineering more difficult. To evaluate the obfuscation resistance performance of SYMGEN and baselines, we reuse SYMLM's obfuscated binary dataset with the same evaluation setup as §VI-A. Specifically, this obfuscation dataset consists of 363 obfuscated binaries from 6 GNU projects (*i.e.*, `findutils`, `coreutils`, `curl`, `less`, `putty`, and `bash`). Following prior works [31], [53], [54], we have used Hikari [46] to obfuscate the binaries three obfuscation options, *i.e.*, bogus control flow (`bcfobf`), control flow flattening (`cffobj`), and instruction substitution (`subobf`). For each obfuscation option, we leverage its default obfuscation configurations. To analyze SYMGEN's performance affected by obfuscation, we also include the original binaries (without obfuscation) in our evaluation dataset.

Table IV presents the performance on obfuscated binaries of SYMGEN, SYMLM and XFL. For simplicity, we also skip

presenting AsmDepictor's performance as it has shown the worse performance in §VI-C. For SymGen, it achieves 0.288, 0.271, and 0.303 average F1 scores on the obfuscated binaries, which shows the 488% better F1 score than SymLM and 777% better than XFL. Compared with the performance of SymGen on the original binaries (w/o obfuscation), SymGen's performance decreases by 4.6% and 10.3% on `bcfobf` and `cffobf` binaries. This degradation is much lower than these in SymLM and XFL. For instance, the performance of XFL decreases by 308.8% in `cffobf`. This demonstrates SymGen's better obfuscation resistance than SymLM and XFL. Moreover, when SymGen achieves the same F1 score on the `subobf` binaries compared as the original ones, SymLM and XFL degrades on these binaries by 12.7% and 256.4%, respectively. We attribute the better performance of SymGen to its specific generalizable learning paradigm with the domain-adapted generative model.

Malware Executables. To further understand SymGen's generalizability, we have used it to predict function names for malware executables. For this, we first collect all Linux malware projects (*e.g.*, backdoors, botnets, infectors, and Mirai) from the prevalent open-source malware repository [70]. We then follow its compilation instructions, resulting in 16 x86-64 binaries, which are further stripped as our test binaries. To apply SymGen, we directly use the model that we have trained *on our own dataset* to generate the function names for all 812 malware binary functions. Surprisingly, SymGen achieves 0.320 precision, 0.381 recall, and 0.348 F1 score. Considering the training dataset of SymGen includes projects of the GNU repository, which have completely different project context from the test malware binaries, this performance is remarkably close to the one that we have reported in §VI. We believe it demonstrates a strong generalizability when SymGen encountered the unknown malware binaries. To understand why SymGen can successfully predict malware function names, we further study and discuss several malware functions in §VII-B.

VII. Further Analysis

In addition to reporting the aggregate statistical performance of SymGen, we study individual cases of the prediction results to understand its prediction errors and generalizability and report our findings in this section.

A. Concrete Cases and Inference Errors

Table V presents the predicted function names of SymGen and our baselines, sampled from our inference results. In this table, both ground truth and SymGen predicted function names are tokenized, consistent with our baselines' outputs, though SymGen produces function names in the same complete form as ground truth. From Table V, we observe that SymGen can accurately infer the function name for some binary functions (*e.g.*, `split_string`), while baselines cannot learn the correct function semantics and generates wrong names. Additionally, SymGen is capable of inferring the core semantics of other functions, generating names that capture essential semantics but may omit or add some details. For example, SymGen correctly identifies the key semantics of `check` and `dir` for the binary function `gc_ide_check_dir` but misses `ide` and the project-specific word `gc`, while our baselines unfortunately fail to grasp the function semantics.

Inference Errors. Despite we have advanced the-state-of-art in inferring binary function names, we still observe inference errors in our results. Beyond the aforementioned issues of omitted or additional words, we identify the other two common error sources, including (*i*) semantic loss due to binary stripping and (*ii*) out-of-vocabulary (OOV) words. In the process of binary stripping, the symbols, such as variable types, are removed. For instance, Figure 7 presents the decompiled code of the function `set_uint32` from the GNU `coreutils-8.32` project. In Figure 7a, the variable `cp` is casted into the type `uint_32` at line 3. This function is named as `set_uint32` to reflect such type casting operation. However, in Figure 7b, the variable type is stripped, leading to a huge semantic gap before and after stripping. SymGen predicts the function name as `assign`, which is treated as an inference error in our evaluations. Another common inference error comes from the out-of-vocabulary words that are specific to projects. For example, in our dataset, the test functions from the `adns` and `dico` projects mostly feature the prefixes `adns_` and `dico_` in their ground truth names. These words are unknown to our model and cause inference errors. One example function is `dico_handle_command`, for which SymGen predicts its name as `do_command`. Although SymGen correctly infers the key semantics of this function, our calculated precision, recall, and F1 score for this function are merely 50.0%, 33.3%, and 40.0%, respectively. While we could identify and remove such prefixes to improve SymGen's performance, we chose not to, aiming to more accurately reflect real-world use cases.

```
1  void set_uint32(char *cp,uint32_t v)
2  {
3      *(uint32_t *)cp = v;
4      return;
5  }
```

(a) Decompiled Code before Binary Stripping

```
1  void FUN_05e4a017(undefined4 *param_1,undefined4
   ↪ param_2)
2  {
3      *param_1 = param_2;
4      return;
5  }
```

(b) Decompiled Code after Binary Stripping

Figure 7: An Example of Semantic Loss in Stripped Binaries. The key semantics of `set_uint32` is preserved in its type (line 3), which is removed in the stripped binary.

B. Generalizability and Effectiveness

Generalizability on Malware Function Name Prediction. We first zoom into individual malware functions, because SymGen achieves impressive generalizability, especially on malware executables (presented in §VI-E). Our analysis identifies that the function names of some functions, highly related to malware, are accurately predicted, *e.g.*, `command_flood` and `get_random_ip`. For instance, Figure 8 presents the decompiled code for the `getRandomPublicIP` function, and Figure 12 (in Appendix §H) presents the source code of the function. This function generates a random public IPv4 address. If a valid range is available in ipState, it increments the fourth part to create and return a new address. Otherwise, it randomly generates an address, ensuring it avoids reserved

Table V: Examples of Inferred Function Names.

Ground Truth	SYMGEN	ASMDEPICTOR	SYMLM	XFL
gc ide check dir	check dir	remove temp	mu init check parse	mu check file
split string	split string	copy string to string	print write file add	mu str set
client session	ssl session load	skip	file write read output	s type check pid
ftp time left accept	get time left	string to	read parse file get	get time read
write error stderr	print error	v print f	print gmp c add	print standard err
read file to buffer	do read file	expand	ipmi monitor ctx	test get padding file
elf type	parse type	expand assignment	p be type	get set ip
gcd odd	gcd odd	mpn copy	u s n gmp	mpn is get
host hash	hash string	png index of number	be p s get	mpz listen

ranges until a valid address is found. Finally, it returns the generated IP address with the last octet set to 0.

After binary stripping, the variable names in the original code have all become meaningless names, like `DAT_0020e389` and `iVar1`, and the function name is replaced with `FUN_041324a`. However, the execution flow of the function has not changed. Surprisingly, SYMGEN precisely infers function semantics from this decompiled code and generates the function name `get_random_ip`. Although the token `public` is missed, we still can get the crucial semantics of random IP generation via its name, which is particularly helpful for malware analysts. Note that, botnets can be used to perform DDoS attacks, steal data, send spam, and allow the attacker to access victim devices [75].

```
1  void get_random_ip (void) {
2    ...
3    if ((DAT_0020e389 == 0)  (DAT_0020e38c == -1)) {
4      iVar1 = rand();
5      DAT_0020e389 = (char)iVar1 + (char)(iVar1 / 0xff);
6      ...
7      DAT_0020e38c = '\0';
8      while (((((((DAT_0020e389 == 0
9                   (DAT_0020e389 == 10))
10                  ...
11                  (0xdf < DAT_0020e389)))))
12     {
13       iVar1 = rand();
14       DAT_0020e389 = (char)iVar1 + (char)(iVar1 / 0xff);
15       ...
16 }
```

Figure 8: The Decompiled Code with SYMGEN Predicted Function Name (Highlighted) for a Botnet Binary Function. Its ground truth name is `getRandomPublicIP`.

Retrieval-augmented Function Name Generation. Our analysis of SYMGEN's outputs also reveals its capability to directly extract key semantic information from the function body, when available. For instance, when we insert the `uint32_t` type at line 3 in Figure 7a, SYMGEN is able to accurately predict the modified function's name as `writ_uint32_t`. To validate if this finding is generalizable, we use the IoT firmware binaries as the test data. For this, we collect 935 test functions from 9 firmware binaries (with ground truth) that are used by existing firmware analysis research [19]. Notably, the firmware function names have many OOV words, *e.g.* IoT-specific terms. Given the IoT binaries, we test SYMGEN using decompiled functions with and without symbols (*i.e.*, variable names and types). We find that the symbols can boost SYMGEN's performance by 258.6%.

In natural language processing, such behavior is defined as retrieval-augmented generation (RAG) [36]. RAG is a process where LLMs can integrate information (*e.g.*, words and phrases) into its output, retrieved from input to enhance the quality of its output. We attribute the emergent RAG capacity of SYMGEN to its two key design advantages. The first advantage lies in SYMGEN's novel generative learning design. Moreover, SYMGEN utilizes decompiled code as binary code representations. The decompiled code can be enhanced by decompilers. Although the stripped binaries can barely have symbols, SYMGEN can be boosted by the outcomes of existing binary symbol recovery frameworks, *e.g.*, variable type and name inference models [10], because of its RAG capacity. This property will benefit other large-scale security analyses [30].

VIII. DISCUSSION

This section outlines the limitations of our study and proposes directions for future research.

Base Model. We employ the pretrained Code Llama as SYMGEN's base model. Given the rapid advancements in the field of generative large language models (LLMs), it is conceivable that future developments could introduce models with superior capabilities in understanding binary code semantics. Furthermore, enhancing SYMGEN with additional model parameters might further augment its performance. Considering SYMGEN's adaptable design to any generative LLMs, exploring alternative base models can be a promising research topic.

Decompiler. In this study, we employ Ghidra, the most popular open-sourced decompiler, to generate decompiled code for SYMGEN. It is important to acknowledge that such decompiler outputs may include errors inherent to the decompilation process. Additionally, the output from different decompilers can vary, potentially introducing biases. Despite SYMGEN's design being decompiler-agnostic, exploring SYMGEN with outputs from alternative decompiler output presents a valuable avenue for future research.

Performance and Inference Errors. The errors introduced by binary stripping and OOV words (§VII-A) remain challenging to resolve due to the missing semantics. Unlike simpler classification tasks, *e.g.*, binary classification, SYMGEN generates each word of function names from the entire English vocabulary, encompassing >170K unique words [43]. This complexity significantly increases the difficulty of accurately predicting function names for SYMGEN. Additionally, SYMGEN's evaluation is conducted at the word level; thus, any inaccuracies in word prediction are rigorously counted in our results, even if those inaccuracies have minimal semantic impact. Despite these rigorous standards, SYMGEN has achieved an

overall F1 score of 0.277, marking a substantial advancement over prior works and a remarkable step towards practicality.

Dataset Quality. Dataset quality is crucial in model training. Besides our efforts in data deduplication and leakage mitigation, low-quality or meaningless function names can also downgrade dataset quality. For instance, some functions are automatically generated by the compiler and linker. Their names do not accurately reflect the function semantics. To address this, we made attempts to manually remove these meaningless functions from our dataset. Moreover, we also found some words appear more frequently than others without carrying much semantics. For example, 4.5% of all function names contain "`set`". These function names usually consist of three to five tokens, and "`set`" represents only 1.2% of the total tokens. We believe the rigorous approach to address this quality issue requires manual annotation. However, it is very time-consuming and relies heavily on the programmer's experience and judgment. Automating the identification of such functions to further enhance dataset quality presents a potential research direction.

Computational Cost. We have also measured the computational cost for training and offline inference. Specifically, the training and reproduction of our baselines, including AsmDepictor, SymLM, and XFL, take 14, 30, and 32 hours per model, respectively, while SymGen costs 41 hours. For offline inference, the time per sample is 0.2, 1, 1, and 3 seconds for AsmDepictor, SymLM, XFL, SymGen. In addition to the time cost, the hardware resources are the same for SymGen and our baselines as we used the same evaluation settings. Note that the training cost is a one-time effort, *i.e.*, the model is trained once but can be used for inference many times. While SymGen is computationally more expensive due to its larger model size, we believe model compression techniques [15], *e.g.*, model quantization and distillation [55], can be applied to reduce this cost.

IX. Conclusion

We have presented SymGen, a novel framework designed for inferring function names in stripped binaries with the new problem formalization. Unlike the classification-based approaches, SymGen is the first to employ a generative learning paradigm to enhance model generalizability for real-world binaries. Our extensive evaluations reveal that SymGen surpasses current state-of-the-art models in effectiveness with better obfuscation resistance. Our ablation studies further validate the effectiveness of SymGen's key components.

Acknowledgment

We appreciate anonymous reviewers for their valuable feedback and comments. This research was supported in part by DARPA award N6600120C4020, ARO award W911NF2110081, and NSF award CNS-2112471. Any opinions, findings, and conclusions expressed herein are the authors' and do not reflect those of the sponsors.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[3] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems*, vol. 34, pp. 27 865–27 876, 2021.

[4] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An {In-Depth} analysis of disassembly on {Full-Scale} x86/x64 binaries," in *25th USENIX security symposium (USENIX security 16)*, 2016, pp. 583–600.

[5] 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," in *33st USENIX Security Symposium (USENIX Security 24)*, 2024.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[7] 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.

[8] J. Cao, W. Zhang, and S.-C. Cheung, "Concerned with data contamination? assessing countermeasures in code language model," *arXiv preprint arXiv:2403.16898*, 2024.

[9] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[10] 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.

[11] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–29, 2021.

[12] X. Chen, R. Aksitov, U. Alon, J. Ren, K. Xiao, P. Yin, S. Prakash, C. Sutton, X. Wang, and D. Zhou, "Universal self-consistency for large language model generation," *arXiv preprint arXiv:2311.17311*, 2023.

[13] Z. Chen, F. Jiang, J. Chen, T. Wang, F. Yu, G. Chen, H. Zhang, J. Liang, C. Zhang, Z. Zhang *et al.*, "Phoenix: Democratizing chatgpt across languages," *arXiv preprint arXiv:2304.10453*, 2023.

[14] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[15] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[16] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *arXiv preprint arXiv:2305.14314*, 2023.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[18] G. Dong, H. Yuan, K. Lu, C. Li, M. Xue, D. Liu, W. Wang, Z. Yuan, C. Zhou, and J. Zhou, "How abilities in large language models are affected by supervised fine-tuning data composition," *arXiv preprint arXiv:2310.05492*, 2023.

[19] B. Feng, A. Mera, and L. Lu, "{P2IM}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[20] H. Gao, S. Cheng, Y. Xue, and W. Zhang, "A lightweight framework for function name reassignment based on large-scale stripped binaries," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.

[21] D. Guo, A. M. Rush, and Y. Kim, "Parameter-efficient transfer learning with diff pruning," *arXiv preprint arXiv:2012.07463*, 2020.

[22] A. Hannun, C. Guo, and L. van der Maaten, "Measuring data leakage in machine-learning models with fisher information," in *Uncertainty in Artificial Intelligence*. PMLR, 2021, pp. 760–770.

[23] 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.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[25] 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.

[26] ——, "loralib: Pytorch implementation of low-rank adaptation (lora)," https://pypi.org/project/loralib/, 2023, accessed: 2024-02-06.

[27] J. Huang, S. S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han, "Large language models can self-improve," *arXiv preprint arXiv:2210.11610*, 2022.

[28] 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.

[29] X. Jin and Z. Lin, "Simllm: Calculating semantic similarity in code summaries using a large language model-based approach," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1376–1399, 2024.

[30] X. Jin, S. Manandhar, K. Kafle, Z. Lin, and A. Nadkarni, "Understanding iot security from a market-scale perspective," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1615–1629.

[31] 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.

[32] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," *arXiv preprint arXiv:2307.10169*, 2023.

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

[34] H. Laurençon, L. Saulnier, T. Wang, C. Akiki, A. Villanova del Moral, T. Le Scao, L. Von Werra, C. Mou, E. González Ponferrada, H. Nguyen *et al.*, "The bigscience roots corpus: A 1.6 tb composite multilingual dataset," *Advances in Neural Information Processing Systems*, vol. 35, pp. 31 809–31 826, 2022.

[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[36] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[37] K. Li, D. Persaud, K. Choudhary, B. DeCost, M. Greenwood, and J. Hattrick-Simpers, "Exploiting redundancy in large materials datasets for efficient machine learning with less data," *Nature Communications*, vol. 14, no. 1, p. 7283, 2023.

[38] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[39] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search." in *NDSS*, 2023.

[40] V. Majdinasab, A. Nikanjam, and F. Khomh, "Trained without my consent: Detecting code inclusion in language models trained on code," *arXiv preprint arXiv:2402.09299*, 2024.

[41] 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.

[42] M. J. Min, Y. Ding, L. Buratti, S. Pujar, G. Kaiser, S. Jana, and B. Ray, "Beyond accuracy: Evaluating self-consistency of code large language models with identitychain," *arXiv preprint arXiv:2310.14053*, 2023.

[43] A. Name, "How many words are in the english language?" https://englishlive.ef.com/en-blog/language-lab/many-words-english-language/, Year, accessed: 2024-02-03.

[44] National Security Agency, "Ghidra," https://ghidra-sre.org/, accessed: 2022-04-21.

[45] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, and M. Conti, "Employing program semantics for malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2591–2604, 2015.

[46] Naville Zhang and Contributors, "Hikari: Llvm obfuscator," https://github.com/HikariObfuscator/Hikari, 2023, accessed: 2024-02-05.

[47] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[48] 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.

[49] T. Parr, "The definitive antlr 4 reference," *The Definitive ANTLR 4 Reference*, pp. 1–326, 2013.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[51] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Annual Computer Security Applications Conference*, 2020, pp. 373–385.

[52] J. Patrick-Evans, M. Dannehl, and J. Kinder, "Xfl: Naming functions in binaries with extreme multi-label learning," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2375–2390.

[53] 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.

[54] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," *arXiv preprint arXiv:2012.08680*, 2020.

[55] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *arXiv preprint arXiv:1802.05668*, 2018.

[56] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[57] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[58] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[59] F. Research, "Llama: Large language model by facebook research," https://github.com/facebookresearch/llama, 2023, accessed: 2024-02-05.

[60] 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.

[61] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 107–119.

[62] J. Su, C. Jiang, X. Jin, Y. Qiao, T. Xiao, H. Ma, R. Wei, Z. Jing, J. Xu, and J. Lin, "Large language models for forecasting and anomaly detection: A systematic literature review," *arXiv preprint arXiv:2402.10350*, 2024.

[63] W. L. Taylor, ""cloze procedure": A new tool for measuring readability," *Journalism quarterly*, vol. 30, no. 4, pp. 415–433, 1953.

[64] T. Theodoridis, T. Grosser, and Z. Su, "Understanding and exploiting optimal function inlining," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 977–989.

[65] tloen, "alpaca-lora: Instruct-tune llama on consumer hardware," https://github.com/tloen/alpaca-lora, 2024, accessed: 2024-02-06.

[66] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[67] P. Tully, S. Vasisht, O. Sardar, and J. Gibble. (2022) Annotating malware disassembly functions with insights from google cloud threat intelligence. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/annotating-malware-disassembly-functions/

[68] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse {Engineers'} processes," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1875–1892.

[69] T. Vu, B. Lester, N. Constant, R. Al-Rfou, and D. Cer, "Spot: Better frozen model adaptation through soft prompt transfer," *arXiv preprint arXiv:2110.07904*, 2021.

[70] vx underground, "Malwaresourcecode," https://github.com/vxunderground/MalwareSourceCode, 2024.

[71] S. Wang, X. Wang, K. Sun, S. Jajodia, H. Wang, and Q. Li, "Graphspd: Graph-based security patch detection with enriched code semantics," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2409–2426.

[72] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language model with self generated instructions," *arXiv preprint arXiv:2212.10560*, 2022.

[73] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[74] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, B. Yin, and X. Hu, "Harnessing the power of llms in practice: A survey on chatgpt and beyond," *arXiv preprint arXiv:2304.13712*, 2023.

[75] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, pp. 1–40, 2017.

[76] E. B. Zaken, S. Ravfogel, and Y. Goldberg, "Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models," *arXiv preprint arXiv:2106.10199*, 2021.

[77] J. Zhai, X. Xu, Y. Shi, G. Tao, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "Cpc: Automatically classifying and propagating natural language comments via program analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1359–1371.

[78] 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)*. IEEE, 2021, pp. 813–832.

[79] T. Zia and U. Zahid, "Long short-term memory recurrent neural network architectures for urdu acoustic modeling," *International Journal of Speech Technology*, vol. 22, pp. 21–30, 2019.

APPENDIX

A. Noises in Function Comments

As stated in §IV-A, we have observed the following five categories of noises during our manual analysis of the sampled source code and comments.

- **Incomplete and Manipulated Comment**. Some comments are in the form of incomplete comments which do not express the full semantics of function semantics. Moreover, some developers incorporate the comment content generated with HTML tags for documentation auto-generation purposes or include URLs for external references, which have changed the original comments created by developers.
- **Wordy Comment**. Some comments come with noisy details that are not expressed in the function bodies, such as the link to some documentation and specific explanations of function arguments, variables, and configurations.
- **Non-English Comment**. In this paper, we focus on the function comments in English. We observe that some developers write comments in their native languages, sometimes mixing these with English within the comments.
- **Comment for Communications**. Some comments are primarily used for communication among developers, rather than just summarizing functionalities. For example, developers use keywords like "TODO" and "FIXME" as the message to inform their co-workers about the next steps.
- **Irrelevant Comment**. We have also observed that some comments are not relevant to the function semantics, which are purely noises. Some of these comments can be the ones that are under construction, meaning that the developers have not finished the comment, therefore, the comments themselves appear to be meaningless.

B. Constructing Abstract Syntax Trees for Source Functions

To construct Abstract Syntax Trees (ASTs) for source code, a parser is needed to interpret the source structure. We build a C code parser based on ANTLR 4 [49] by adapting a C language grammar file[4]. Provided with source code, the parser will produces an AST representing the code's structure. Each explicitly defined grammar rule in the grammar file corresponds to a node in the AST. For instance, the following rule defines the grammar of function declaration.

```
1 functionDefinition:
2     declarationSpecifiers? declarator
      ↪  declarationList? compoundStatement;
```

In this rule, *declarationSpecifiers*, *declarator*, *declarationList*, and *compoundStatement* represent the function's return type, name, input arguments, and body, respectively. Each of these components is further defined by its own set of rules. If a function declaration appears in the source code, a node of type *functionDefinition* will be created in the AST by parser, with four child nodes representing these attributes.

After generating the AST, we can perform a depth-first search (DFS) to traverse each node, aiming to collect all function bodies in the source code. The algorithm is presented in algorithm 2. The traversal begins at the AST root using function `CollectFunctions` (lines 8 to 12). As each node is visited, the code checks if the node's type is *functionDefinition*. If true, this indicates that the node corresponds to a function declaration, allowing us to collect its content, which represents the function body (lines 2–7).

C. Prompts for Function Summary Generation and Function Name Inference

The choice of prompt is especially critical to the results of the model. For generating function summary and predicting function name, we tried many different prompts on a small dataset and finally chose the prompts with best performance. For function summary task, Figure 9a presents the prompt we used. For predicting function name, the prompts with assembly code and decompiled code as input are shown in Figure 9b and Figure 9c. As mentioned in §IV-C, the original artificial function name in decompiled code is replaced by the [MASK]

[4] https://github.com/antlr/grammars-v4/blob/master/c/C.g4

Algorithm 2: Collect Functions from AST

```
initialize funcMap := {}
Function CollectFunctions(node):
    if node.type == functionDefinition then
        funcMap[node.name] := (node.args, node.content)
    for each child in node.Children do
        CollectFunctions(child)
    return
Function DFS():
    root := getASTRoot()
    CollectFunctions(root)
    return
```

Table VI: Source Projects and Binaries in Our Dataset

Project	# Binaries	Project	# Binaries
openssl-3.0.6	2128	libiconv-1.17	48
coreutils-8.32	1848	grep-3.8	48
ncurses-6.3	1440	gmp-6.2.1	40
readline-8.2	568	libidn2-2.3.4	32
bash-5.2	472	tar-1.34	32
adns-1.6.0	448	datamash-1.8	32
gettext-0.21	444	curl-7.86.0	32
inetutils-2.4	400	gss-1.0.4	32
binutils-2.39	352	poke-2.4	32
dico-2.11	276	libpng-1.6.39	32
gawk-5.2.1	240	gzip-1.12	16
libredwg-0.12	192	libmicrohttpd-0.9.75	16
mailutils-3.8	180	libunistring-1.0	16
wget2-2.0.1	136	cflow-1.7	16
freeipmi-1.6.10	112	libtool-2.4.7	16
diffutils-3.8	96	units-2.22	16
texinfo-7.0	54	-	-

token without altering any other portion of the decompiled code (*e.g.*, function body and function parameters.)

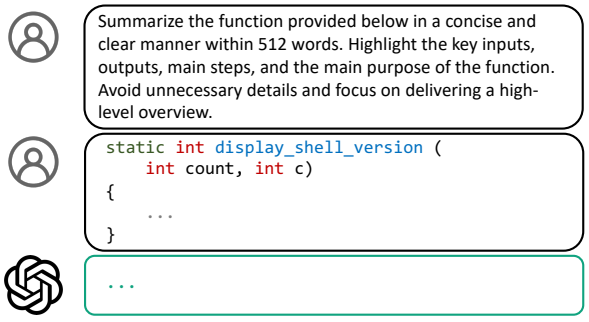
D. Source Project and Binaries in SymGen *dataset*

Table VI presents the 33 open-source projects that we used to build our dataset, which contains 9,842 unique binaries in total. We have chosen well-known GNU projects, including `bash`, `wget`, and so on. `bash` is a Unix shell and command language developed as part of the GNU Project. It provides a command-line interface for users to interact with their operating system, offering powerful scripting capabilities and extensive support for automating tasks. `wget` is a command-line utility for downloading files from the web. It supports downloading via HTTP, HTTPS, and FTP protocols, and can recursively download entire directories with ease. These projects are widely used and highly representative.

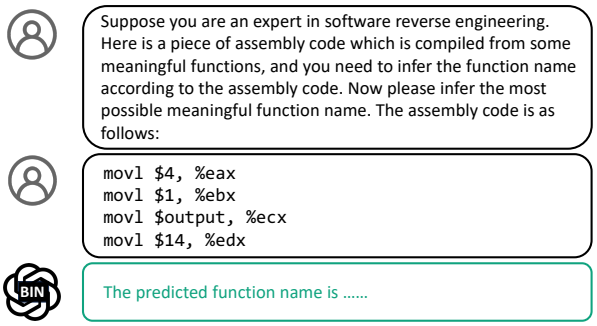
E. Substitution in Function Body

In the decompiled code, the callee function names will affect the identification of the same function body. Due to the lack of debuging information, the function names in the function call sentences is the combination of `FUN` and its address. Figure 10a presents the function `add_intrinsic` before preprocessing.

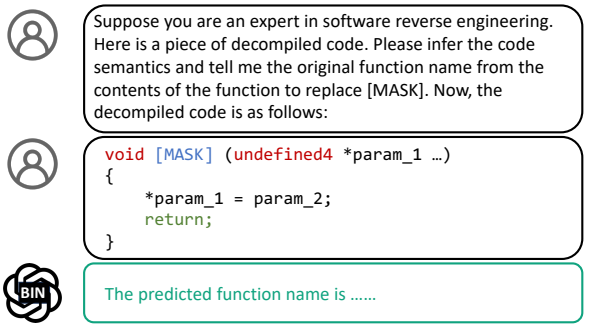
Using ANTLR 4 to build the AST (Appendix §B), then we substitute these callee function names with consistent names.



(a) Prompt for Function Summary Generation



(b) Prompt of Assembly Code as Input



(c) Prompt of Decompiled Code as Input

Figure 9: Prompts for Summary Generation and Function Name Prediction.

Figure 10b presents the function after preprocessing. New function names are based on the order of appearance.

F. Illustration of Our Evaluation Metrics and CodeWordNet

To help readers understand how our evaluation metrics (defined in §VI-A) work, we use the function `ftp_timeleft_accept` in forth line of Table V as an example to demonstrate how we calculate the scores. First, we will divide the function name into a series of tokens. For example, `ftp_timeleft_accept` is divided into `ftp`, `time`, `left`, `accept` and our predicted name is divided into `get`, `time`, `left`. Then we calculate TP, FP and FN. True positive (TP) refers to a result in binary classification where the model correctly identifies a positive instance as positive. False positive (FP) refers to the incorrect identification

```
void [MASK](undefined8 param_1,undefined8
↪  param_2)
{
  undefined8 uVar1;

  uVar1 = FUN_00000c4b(param_1);
  uVar1 = FUN_00003016(uVar1);
  FUN_00003086(uVar1,param_2);
  FUN_00003086(uVar1,0);
  return;
}
```

(a) The Decompiled Code of `add_intrinsic`

```
void [MASK](undefined8 param_1,undefined8
↪  param_2)

{
  undefined8 uVar1;

  uVar1 = FUN_0(param_1);
  uVar1 = FUN_1(uVar1);
  FUN_2(uVar1,param_2);
  FUN_2(uVar1,0);
  return;
}
```

(b) The Decompiled Code of `add_intrinsic` after Processing

Figure 10: Comparison of Decompiled Code Before and After Processing

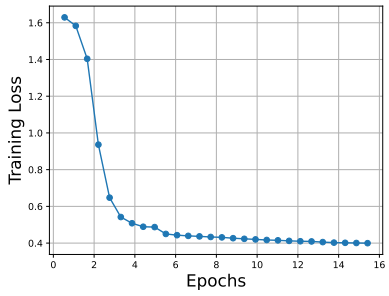


Figure 11: Training Loss across Epochs

of something as belonging to a certain category, while false negative (FN) is the failure to identify something as belonging to that category when it actually does. In this example, token `time` and `left` belong to true positive and `get` belongs to false positive. The predicted name missed `ftp` and `accept`, both of which belong to false negatives. So the precision, recall and F1 score of this function are 0.67, 0.50 and 0.57. We have used the CodeWordNet module of SYMLM [31] to precisely match semantically similar words. Specifically, CodeWordNet is trained on the word embedding models which can generate the close embeddings for morphological words, *e.g.*, synonyms. Using CodeWordNet, SYMGEN can precisely calculate the scores based on its embedding models.

G. Training Loss across Epoches

When applying parameter-efficient training, we observe the early-stop effect during training. Figure 11 presents the training loss of SYMGEN across the training epochs. The loss reaches a stable status at 6 epochs, while our default setting is 16.

```
in_addr_t getRandomPublicIP()
{
    if(ipState[1] > 0 && ipState[4] < 255)
    {
            ipState[4]++;
            char ip[16] = {0};
            szprintf(ip, "%d.%d.%d.%d", ipState[1],
            ↪  ipState[2], ipState[3], ipState[4]);
            return inet_addr(ip);
    }

    ipState[1] = rand() % 255;
    ipState[2] = rand() % 255;
    ipState[3] = rand() % 255;
    ipState[4] = 0;
    while(
            (ipState[1] == 0)
            (ipState[1] == 10)
            (ipState[1] == 100 && (ipState[2] >= 64
            ↪  && ipState[2] <= 127))
            (ipState[1] == 127)
            (ipState[1] == 169 && ipState[2] == 254)
            (ipState[1] == 172 && (ipState[2] <= 16
            ↪  && ipState[2] <= 31))
            (ipState[1] == 192 && ipState[2] == 0 &&
            ↪  ipState[3] == 2)
            (ipState[1] == 192 && ipState[2] == 88
            ↪  && ipState[3] == 99)
            (ipState[1] == 192 && ipState[2] == 168)
            (ipState[1] == 198 && (ipState[2] == 18
            ↪  ipState[2] == 19))
            (ipState[1] == 198 && ipState[2] == 51
            ↪  && ipState[3] == 100)
            (ipState[1] == 203 && ipState[2] == 0 &&
            ↪  ipState[3] == 113)
            (ipState[1] >= 224)
    )
    {
            ipState[1] = rand() % 255;
            ipState[2] = rand() % 255;
            ipState[3] = rand() % 255;
    }

    char ip[16] = {0};
    szprintf(ip, "%d.%d.%d.0", ipState[1],
    ↪  ipState[2], ipState[3]);
    return inet_addr(ip);
}
```

Figure 12: The Source Code of `getRandomPublicIP`

H. Source Code of Malware Function

Figure 12 presents the source code of function `getRandomPublicIP` in botnet.